

PARTITIONING ARTIFICIAL NEURAL NETWORKS ONTO COARSE GRANULAR
PARALLEL SYSTEMS

By
KARL S. GUGEL

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1993

ACKNOWLEDGEMENTS

Gratitude is expressed to Dr. Bert “Papper” DeVries, Dr. Jyh Ming Kuo and Dr. Jose C. Principe for their patience and time in answering my thousands of questions regarding retrieval, backpropagation and the general function of static and dynamic neural classifiers. Similarly, appreciation is extended to Dr. Michel Lynch for his guidance and suggestions regarding new computer hardware designs and architectures.

In addition, I would like to thank the following masters’ students for their contributions to this research: Todd Hornquist, for his neural net software development on the multiple DSP56000 board, Suresh Venkumahanti for his CNEL Machine hardware design/ simulations (and cancan dancing skills), and John Robinson for CNEL Machine board construction and assembly coding.

Special consideration is also given to Dr. Armando “Captain” Barreto for his dedication to engineering perfection, Elton John DJ skills and ability to set a very high Ph.D. standard in the lab. An honorable mention is presented to my brother-in-law, Russ Walters, for his system administration skills and nuptial agreement with Yukako Walters.

My father, mother, sister and wife should also receive recognition for their steadfast support and constant questioning of “when do you plan to finish school?” Without this, I would have undoubtedly continued my Ph.D. through eternity.

Finally, I praise Dr. Jose C. Principe’s style, editing skills and ability to keep everyone in the lab focused. Gratitude is also extended to Mrs. Principe for keeping Dr. P. in line.

TABLE OF CONTENTS

	<u>Page</u>
ACKNOWLEDGEMENTS.....	ii
ABSTRACT.....	vi
 CHAPTER	
1 THE HIGH COMPUTATIONAL BANDWIDTH REQUIREMENT OF ARTIFICIAL NEURAL NETWORKS.....	1
1.1 Introduction.....	1
1.2 The Inherent Neural Computation Problem.....	1
1.3 Faster Computing Engines to Speed Learning.....	3
1.4 The Goals of this Research.....	3
1.5 Scope of the Dissertation.....	5
1.6 Content.....	6
 2 THE MULTI-LAYER PERCEPTRON: A STATIC ARTIFICIAL NEURAL NETWORK.....	 8
2.1 Introduction.....	8
2.2 Network Topology and Notation.....	9
2.3 Multi-layer Perceptron Retrieval Mode.....	11
2.4 Multi-layer Perceptron Learning.....	13
2.5 Parallelism and Pipeline Ability in the Backpropagation Algorithm.....	15
 3 IMPLEMENTATIONS IN ARTIFICIAL NEURAL NET SYSTEMS.....	 19
3.1 Introduction.....	19
3.2 Analog Domain.....	19
3.3 A Mixture of both Worlds.....	21
3.4 Digital Realizations.....	22
3.5 Neuro Turbo: a DSP-based Approach to Neural Computing.....	24
3.6 S.Y. Kung's Systolic Array Neural Network Implementations.....	38
3.7 Chapter Summary.....	50

4	THE GAMMA MODEL: A DYNAMIC NEURAL NETWORK.....	55
4.1	Introduction.....	55
4.2	Topology and Retrieval.....	57
4.3	Backpropagation in Recurrent Networks.....	61
4.4	Parallelism in the Focused Gamma Network.....	63
4.5	Partitioning Locally Recurrent Neural Networks.....	64
5	THE APPLICATION OF ATPD TO A GIVEN ALGORITHM AND ARCHITECTURE.....	69
5.1	Introduction.....	69
5.2	General Purpose Serial Computer.....	70
5.3	Expanding the Serial Computer (Retrieval Mode Only) with Pipelining.....	75
5.4	ATPD Analysis of Backpropagation.....	81
5.5	Adapting the ATPD Backpropagation Equations to Different Architectures.....	85
5.6	ATPD Analysis of Neuro Turbo's DSP System and Kung's Systolic Arrays.....	86
5.7	ATPD Analysis of the Focused Gamma Model.....	96
5.8	Conclusions.....	99
6	MULTIPLE DSP ARCHITECTURES.....	101
6.1	Introduction.....	101
6.2	Narrowing the Spectrum of Available Neural Technologies.....	102
6.3	Three Basic DSP Architectures.....	105
6.4	ATPD Analysis of Three Basic DSP Architectures.....	107
6.5	Creating an Interactive Environment for the Computation Time Formulas.....	111
6.6	Results using the Coarse Parallel System Estimator Software.....	117
6.7	The CNEL Machine.....	127
6.8	Chapter Summary.....	131
7	WORKSTATION CLUSTERING.....	133
7.1	Introduction.....	133
7.2	Clustering NeXT Machines.....	135
7.3	Initiating PVM and Example Code.....	137
8	MULTIPLE DSP SYSTEM AND CLUSTERING RESULTS.....	144
8.1	Introduction.....	144
8.2	Estimating System Primitives.....	145
8.3	Multi-layer Perceptron Results.....	148

8.4 Focused Gamma Net Results.....	157
8.5 Chapter Summary.....	163
9 SUMMARY AND FUTURE WORK.....	169
9.1 Summary of Research.....	169
9.2 Future Work.....	178
REFERENCES.....	181
BIOGRAPHICAL SKETCH.....	185

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

PARTITIONING ARTIFICIAL NEURAL NETWORKS ONTO COARSE GRANULAR
PARALLEL SYSTEMS

By

Karl S. Gugel

December 1993

Chairman: Jose C. Principe
Major Department: Electrical Engineering

This dissertation presents a methodology for comparing and evaluating parallel computer architectures (and associated partitioning schemes) executing artificial neural networks. The methodology consists of analyzing the algorithm for parallelism, partitioning the equations on a coarse parallel machine and developing a cost function for the total computation time required by the parallel realization. The tool used to create the cost function is termed algorithmic timing parameter decomposition (ATPD). It essentially breaks a complex process into sums and products of system dependent primitives. From these primitive sums, one can observe the contributing elements comprising the total computation time of the algorithm. This information can be employed for prediction purposes or to redesign and optimize the parallel architecture.

Specifically, a multi-layer perceptron and the locally recurrent focused gamma network were partitioned across two existing parallel systems, and their performances were analyzed with ATPD. The first system is a NeXT Machine prototype board containing four Motorola DSP56000s and the second a cluster of NeXT workstations. Following this analysis, a hybrid system was proposed for accelerated neural processing. The proposed system is known as the CNEL Machine and is a dual bus architecture comprised of several Texas Instruments TMS320C31 floating point digital signal processors (DSPs).

CHAPTER 1

THE HIGH COMPUTATIONAL BANDWIDTH REQUIREMENT OF ARTIFICIAL NEURAL NETWORKS

1.1 Introduction

My interest in artificial neural networks (ANNs) was first aroused upon hearing that a new field of information processing was developing that attempted to emulate the basic physical and psychological properties of the human brain. Proposed were systems based on massive numbers of elementary processors, high connectivity between these simple processors, extensive parallelism and a processing element that employed a nonlinear decision mechanism. Upon starting research in 1991 in the Computational Neuro Engineering Lab (CNEL), this technology was known to be relatively young with few real world applications. However, over the last few years, the number of ANN-based systems and related research has greatly increased. Presently, neural net approaches to solving problems can be found in a wide range of fields such as speech recognition, pattern classification, alphanumeric character recognition, economic forecasting and system control [1][2].

1.2 The Inherent Neural Computation Problem

With this boom in artificial neural network related research, comes many new problems associated with an algorithm's physical execution time, i.e., how long it takes to get an answer and/or to test a proposed solution. For example, it is not uncommon for many researchers using work stations as neural computation engines to wait several days for an acceptable output. Compounding this problem is the fact that there are very few theoretical tools available. The field is considered very experimental; the process of

optimizing an algorithm may require several experiments or adaptations. Thus any delay in obtaining an answer hinders the research process and slows the emergence of new real world applications. However before this problem is discussed further, a brief explanation of what comprises a neural network should first proceed.

An artificial neural network, or ANN, is an adaptive nonlinear distributed classifier. The classifier portion of the phrase derives from the fact that ANNs are used to classify from one dimensional space to another. "Distributed" implies that the information is distributed or spread out over several elements. No single node contains all the classification information. "Nonlinear" corresponds to the type of transfer function used at the processing element (PE), and "adaptive" suggests that the system can be trained. Examples of the nonlinear transform and the property of distribution are presented in the next chapter.

In ANNs, how one trains or adapts the network is termed the "learning mechanism." Depending on the type of learning mechanism, two distinct categories arise. They are supervised and unsupervised learning. Under supervised learning, a network is presented an input and the generated output is compared to a known or desired value. In the case of unsupervised learning, the network is given an input and then it is allowed to self-organize according to principles embodied in the interactions. This dissertation addresses the problems associated with supervised learning, which is the most popular learning mechanism currently used in neural research.

Supervised learning, as mentioned earlier, presents an input vector to the network for classification and then compares the resulting output to a desired vector. The comparison to a desired signal generates error terms, which are then used to incrementally adapt the weights comprising the network. Several hundred thousand incremental adaptations (using one pair of input/desired training vectors) may occur before the resulting output error is satisfactorily minimized or convergence occurs. Thus this is an extremely

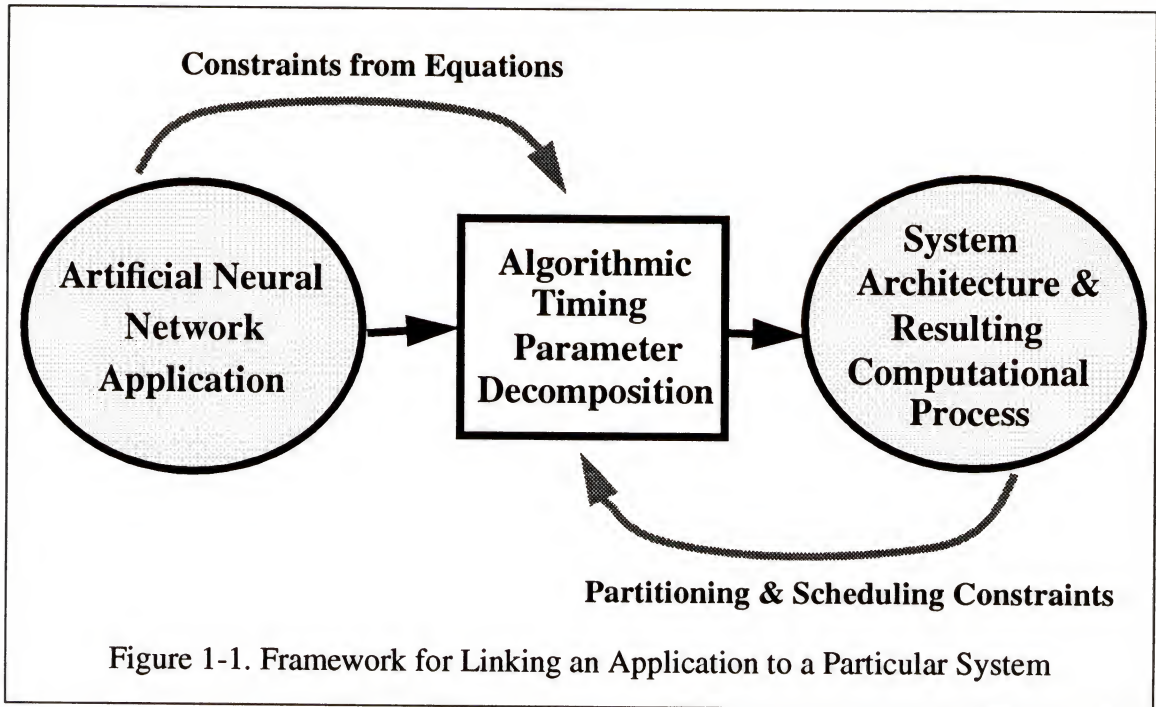
computationally intensive process. The problem further increases in complexity as the size of the network increases. For example, if the number of processing units (PEs) per layer is N , the corresponding weight layers will be N^2 for a fully connected network. Thus for increasingly sized networks, the activations (N) grow linearly while the weights grow as a quadratic function (N^2).

1.3 Faster Computing Engines to Speed Learning

Because of this high computational bandwidth requirement, many researchers have turned to parallel processing for improvements in processing [3][4][5][6]. These systems have been based on analog hardware, digital realizations and mixtures of both. Analog examples are optical computers, multiple OP amp systems and CMOS chips, which sum current using traditional analog circuit techniques. Digital realizations span fine, coarse and medium granular parallel systems. A few examples are the Connection Machine, systolic arrays, digital signal processor (DSP) rings and a cluster of workstations. The question now broadens to what type of parallel system is required for a given application/neural research area and how should the original algorithm be partitioned across the multi-processor system for efficient parallel processing.

1.4 The Goals of this Research

Specifically, the goal of this dissertation is to present a methodology to compare the speed of various machines (serial and/or parallel) when executing neural processing algorithms, the desire being actually to quantify the efficiency of a particular partitioning scheme on a given architecture and also help specify architectural design trade-offs. Both static and dynamic neural nets will be partitioned across a cluster of processors and analyzed in terms of performance and efficiency. To date, no literature has been published that discusses the partitioning of recurrent neural networks. Thus another goal of this research was to provide an efficient (high degree of parallelism) partitioning scheme for networks



that are locally recurrent, in particular the class known as dynamic neural nets (static networks with short term memory structures).

In either of the static or dynamic network partitioning cases, it will be necessary to identify the elements or primitives of the design that contributed to the overall algorithm computation time. A valuable by-product of this analysis is that weaknesses in the design can now be observed/located and minimized in future designs.

The method of analysis used to link a particular system to a given neural application must take into account both the equations in the application and the relationship between the algorithm and the architecture (partitioning and scheduling schemes). See Figure 1-1. The analysis tool selected is termed algorithmic timing parameter decomposition (ATPD). ATPD weighs the specifics of the algorithm under analysis, various architectures, processing schedules and corresponding partitioning schemes available as solutions. Using ATPD

and the procedure outlined in this dissertation, one should be able to estimate the performance of their neural application on a particular neural processor. In addition to performance evaluation, ATPD will also be shown to be an excellent tool for locating the strong points and weaknesses in the system design. Hence, this information can be used to improve the neural computation process.

1.5 Scope of the Dissertation

As in other areas of computer processing, there is usually a trade-off between speed and price. This research focuses primarily in the area of low cost, high-performance systems that would be affordable by small companies, university labs, and/or individuals. Hardware costs therefore dictate that the domain of possible system solutions be limited to architectures based on either a single processor or a few processors (coarse grain parallel systems). Due to the recent availability of inexpensive microprocessors and digital signal processing (DSP) chips, it is economically sound to increase computing speed by using several of these types of processors in one unit known as a back-end system. Another possible solution is to link several existing serial machines (i.e., workstations in a lab) as a single parallel cluster. Both types of parallel realizations are critiqued in this dissertation and, actual execution times are given for comparison to earlier estimated values.

The neural algorithms selected for analysis and partitioned were the multi-layer perceptron (MLP) and the focused gamma model. The multi-layer perceptron is referred to as a static classifier and is the most common artificial neural network model employed in neural research today. The focused gamma network is a dynamic neural network containing local feedback in the input layer processing elements (PEs). This network can be thought of as an extension of the multi-layer perceptron in that it contains a short-term memory structure in the input layer.

1.6 Content

The second chapter in this dissertation is dedicated to introducing the basic static neural network known as the multi-layer perceptron (MLP). This chapter illustrates both the equations associated with an MLP and presents the network terminology required by the cost analysis tool, ATPD. Chapter 2 also analyzes the neural algorithm equations for parallelism and pipeline ability.

Chapter 3 is meant to be a bibliographic review that introduces several different hardware neural processing systems. Two particular systems are reviewed in detail to illustrate how a multi-layer perceptron can be partitioned across a parallel system. In addition, this chapter discusses the pros and cons of a particular neural technology.

The next chapter, 4, presents a dynamic neural network known as the gamma model. The static neural model from Chapter 2 is expanded by adding a short-term memory structure. This structure is termed a gamma kernel. The equations for the locally recurrent network are presented and again analyzed for parallelism. Upon completion of this, a processing scheme is presented for partitioning locally recurrent networks onto coarse grain parallel systems.

Chapter 5 then illustrates how to apply algorithmic timing parameter decomposition to the multi-layer perceptron in Chapter 2 and to the gamma model found in Chapter 4. General total computation time models are derived to estimate how fast an algorithm will execute on a given architecture. From the general models, specific multi-layer perceptron computation time models are then derived for a DSP ring and systolic array introduced in Chapter 3.

Chapter 6 expands the general neural timing models developed in Chapter 5 to include models for parallel systems based on digital signal processors (DSPs). Three basic DSP architectures are modeled and critiqued for performance versus cost. After analysis of

the three systems, one particular architecture was selected for fabrication in the Computational NeuroEngineering Lab (CNEL). This system, termed the CNEL Machine, is currently being constructed, and a working prototype is expected by the year's end. The specifics of this system are described in Chapter 6.

Chapter 7 shifts to another type of coarse parallel solution that consists of clustering workstations. This is performed using a shareware program known as the Parallel Virtual Machine or PVM for short. The function and usage of this software is presented in detail. Chapter 8 then illustrates the results obtained from both a multiple DSP system and work station clustering. Actual computation times are compared to estimations derived from the models presented in Chapter 9. Chapter 8 also presents a summary of this research, conclusions and the future direction of neural-computer engineering related research in the lab.

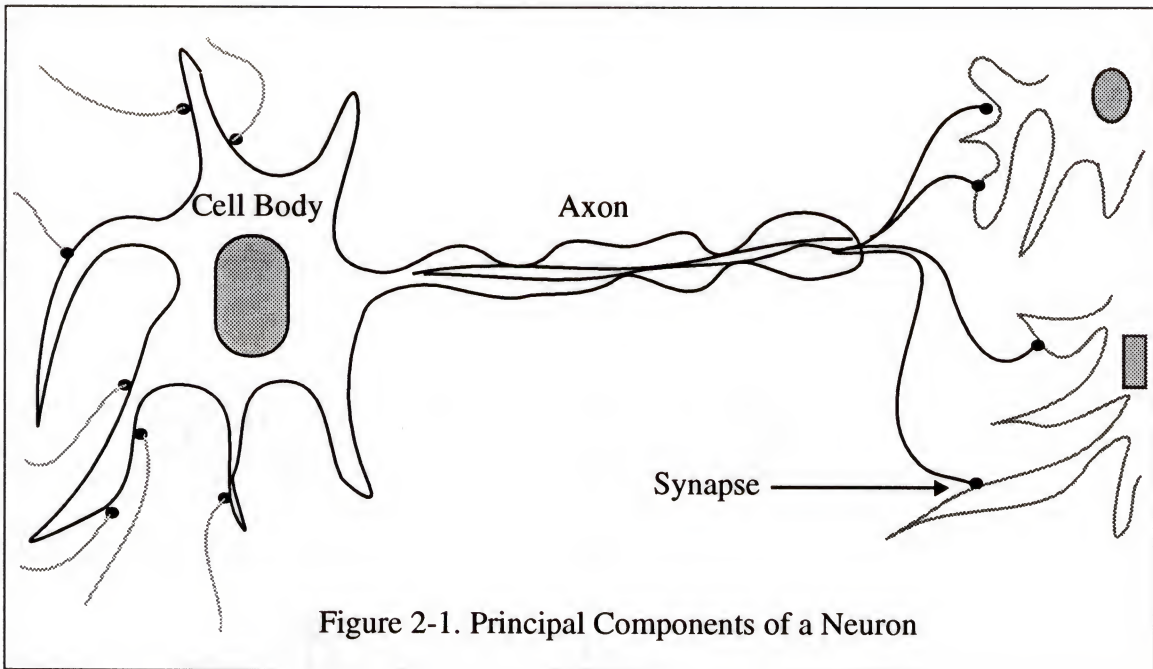
CHAPTER 2

THE MULTI-LAYER PERCEPTRON: A STATIC ARTIFICIAL NEURAL NETWORK

2.1 Introduction

Artificial neural networks are a branch of numerical processing born out of both signal processing and biological research on the human brain. ANNs are comprised of simple processing elements (PEs), which can number from a few to several hundred thousand. Similarly, but on a much greater scale, the human brain is composed of approximately one trillion principal building blocks/nerve cells known as neurons [7][8][9]. These neurons are connected to one another via axons, synapses and dendrites. See Figure 2-1. The axon acts like an output, feeding synapses, which connect to dendrites (inputs). A synapse is a sort of chemical resistor that is used to regulate activity received from neighboring axons. Hence information processing in the neuron is performed by integrating (summing) information received through the dendrites and comparing it to an internal threshold value. If this information is greater than the threshold, the neuron will fire and an impulse will be generated along the axon. Neighboring neurons weight this information (via their local synapses) and the process repeats.

In addition to this large number of neurons, it is estimated that there are 1000 connections per neuron or 10^{15} total connections in the human brain. Thus there is enormous parallelism and extremely high connectivity. Artificial neural nets attempt to reproduce some of these functions and qualities but on a much smaller, simpler scale. In Figure 2-2, the inputs are shown as x_1 - x_6 while synapses now become weights w_1 - w_6 . The inputs are therefore multiplied by the corresponding weights and summed at the processing element (PE). This sum is then passed through a nonlinear transform to yield a single output



referred to as an “activation.” This process is mildly similar to a neuron firing in the human brain.

2.2 Network Topology and Notation

Combining several of the processing elements (PEs) shown in Figure 2-2, it is possible to build networks that contain several layers of PEs. See Figure 2-3. This is referred to as a two layer multi-perceptron neural network [10][11][12][13][14]. The PEs that generate activations (network outputs) y_1 - y_N are referred to as the output layer PEs. The layer of PEs connected to the inputs via weights is known as the hidden layer. A layer of weights is associated with the PE layer immediately to the right of it. Hence the two weight layers are referred to as the output layer weights and the hidden layer weights. The notation used to define a particular input, PE, activation, or weight shown in Figure 2-3 is described in Table 2-1. The number of inputs or PEs in a given k layer is denoted N_k . Uppercase characters are therefore used to denote a total quantity while lower case characters are used for

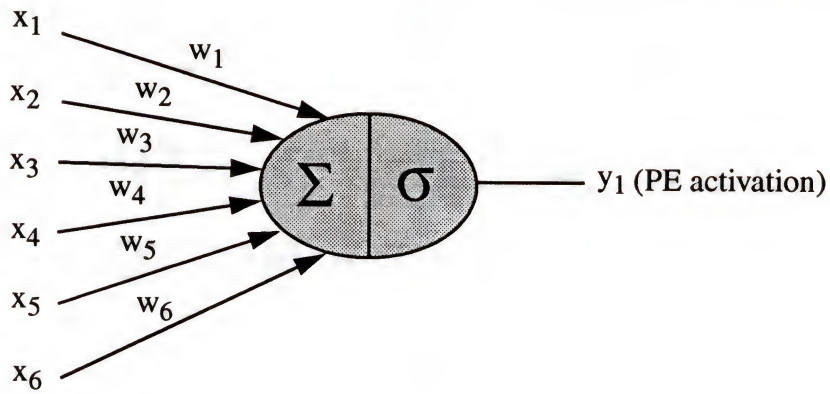
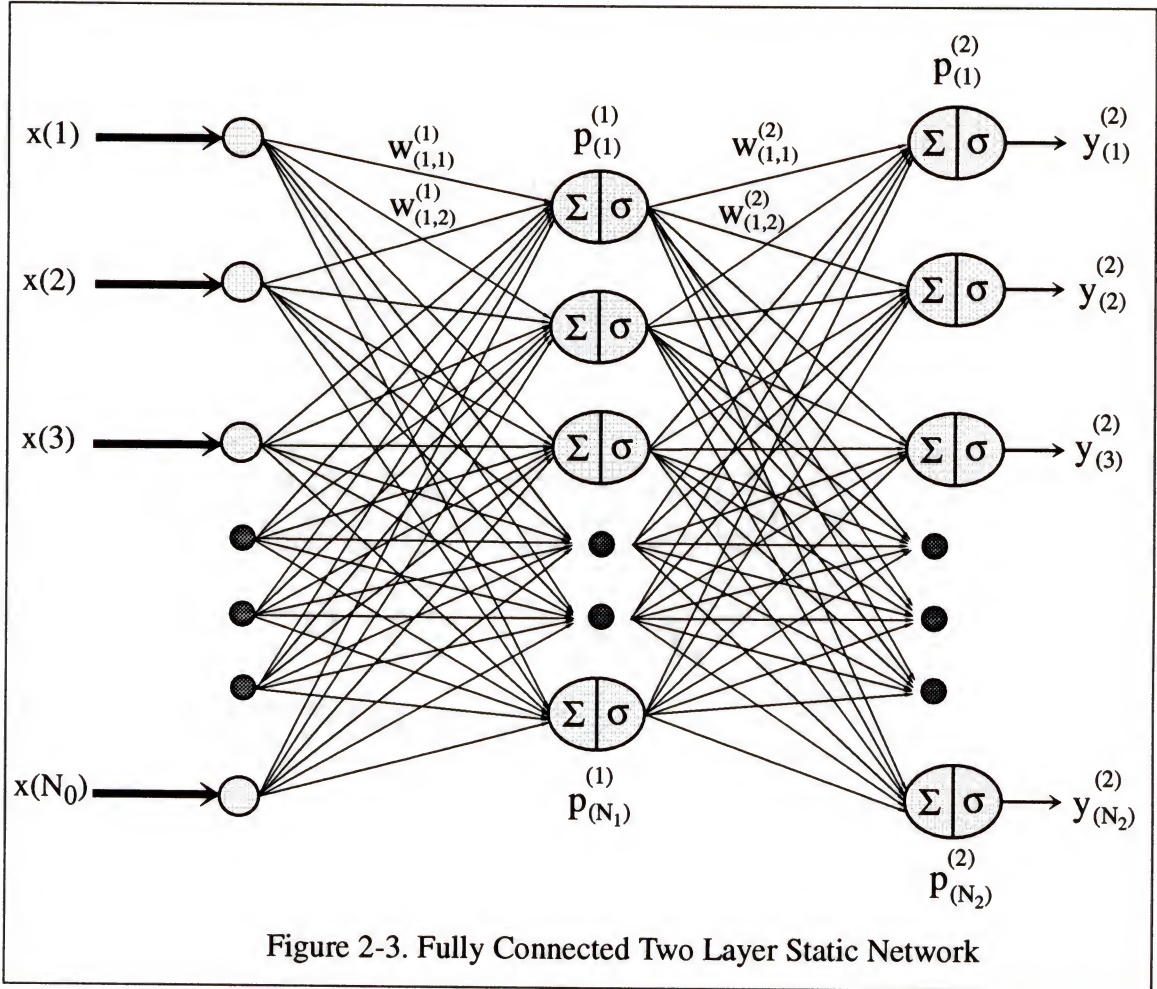


Figure 2-2. The Neural Net Processing Element (PE)

Table 2-1. Network Variable Notation

Variable	Function	Additional Comments
x_j	input vector	j is the input index.
y_i^k	activation	k is the layer index, i in the index for a particular PE output.
p_i^k	processing element	k and i are layer and PE indices, respectively.
w_{ij}^k	weight	k is the layer corresponding to the weight's termination. j is originating index, i is terminating.

indexing a particular PE, input, activation or weight. Specifically, superscripts denote a particular layer and subscripts are used for indexing inside a layer. Hence, in Figure 2-3, the total number of inputs is shown as N_0 . This quantity usually corresponds to the input pattern that will be presented to the network. The number of output values is shown as N_2 , and this corresponds normally to the number of classifications the net will be required to



decipher. The hidden layer N_1 can be arbitrarily set to any value between the input and output layers.

2.3 Multi-layer Perceptron Retrieval Mode

Once a multi-layer perceptron has been trained, as mentioned in the first chapter, the weights are fixed and it is used in a forward propagation mode called “retrieval.” During retrieval, also referred to as “operation mode”, an input vector is presented to the left-most layer PEs (via the weights) and then spatially propagated from the left to the right until an output vector is obtained. In Figure 2-2, the ANN is in essence mapping an N_0 dimensional

input vector to an N_2 dimensional output vector (classification). In the equation that governs retrieval (2-1), $\sigma()$ is a point-wise nonlinear transform and y_i is the computed activation; k corresponds to a particular layer as before, with i being the weight's terminating index and j the originating index. When the activations are computed for the $k=1$ layer $y_k^0 = x_j$, the input vector. Equation (2-1) can also be rewritten in terms of a dot product variable called the net. See (2-2). This is required later in the learning equations described in section 2.4.

$$y_i^k = \sigma \left(\sum_{j=1}^{N_{k-1}} (w_{ij}^k y_j^{k-1}) \right) \quad (2-1)$$

$$y_i^k = \sigma(u_i^k) \quad \text{where,} \quad u_i^k = \sum_{j=1}^{N_{k-1}} (w_{ij}^k y_j^{k-1}) \quad (2-2)$$

2.3.1 Parallelism in Retrieval Mode

In order adequately to determine the levels of parallelism in an algorithm or process, one must first determine the inherent dependencies in the algorithm [15]. S.Y. Kung from Princeton does this by mapping the dependencies of a particular algorithm to a directed graph. This is presented in Chapter 3, section 3.6. Another, simpler procedure is to write the equations composing an algorithm in a sum of products form as in equation (2-1). For example, by careful inspection, it can be seen that the iterative nature of equation (2-1) yields a dependency between a particular layer's output with respect to the previous layer's output. Thus in Figure 2-3 there cannot be any horizontal parallelism.

However, pipelining can be implemented in this flow direction and is discussed in section 2.3.2. Equation (2-1) does allow for vertical parallelism in a particular layer. In equation (2-1), a particular i output calculation is independent of another i calculation (i.e., PE_i , PE_{i+1} , $PE_{i-1}, \dots$). If the PEs in a given layer can simultaneously access the j inputs and

weights, they can process in parallel because they are autonomous units. Hence the i output computations can all be performed in parallel for a particular k layer. This behavior suggests that the multi-layer perceptron net shown in Figure 2-3 may be implemented as a single instruction multiple data architecture [16]. Under this assumption, the PEs in a layer would all have the same instruction sequences but would operate on different data streams.

2.3.2 Pipeline Ability

During retrieval mode, pipelining exists between the layers. In a feed forward network the PEs can concurrently compute both within a given layer and with other PEs in other network layers. For example, all the layers of PEs could perform the parallel PE summation/nonlinear transformation in the first portion of a cycle and then pass the PE outputs to the next layer in the last portion of the cycle. In this scenario the k^{th} layer would always be one state behind the $k-1^{\text{th}}$ layer. However with this full parallelism inside a layer and pipelining between layers, it is possible for a system to have a maximum throughput of one output vector per an individual PE's calculation/transfer cycle. The architectural and scheduling requirements for full vertical parallelism and horizontal pipelining in the retrieval mode will be discussed further in Chapter 5.

2.4 Multi-layer Perceptron Learning

An artificial neural network can be thought of as being composed of three fundamental descriptors: a connection topology, transfer function and learning law [13]. While the first two primal elements previously discussed are simple to grasp, the learning paradigm for a net may appear more complex at first glance. Therefore, greater care and detail will be presented in this section regarding this subject.

The learning law for a neural net defines exactly how the net's weights are adapted or changed in response to a given input/output vector pair. A training set consists of a set of input vectors and a corresponding set of desired or target output vectors. The difficulty

begins on how we calculate the error between the net's actual output and the desired vector and how that error is propagated back to the k-1, k-2,... layers. Presently, the most widely used error criteria by neural researchers is the Sum of Squared Errors (from the Least Mean Square algorithm) error definition [11] [13] [8]. Here the error is calculated as the sum of squares of the desired outputs minus the target value. Although the error can be computed in other ways, e.g. the sum of absolute values of individual errors, the sum of squared error definition is commonly used because its derivative is easily computed [10]. This error estimate at the output is then backpropagated to the hidden layers through a learning paradigm known as backpropagation. From this error estimation the weights can be incrementally adapted. The equations governing this process are shown in (2-3) through (2-7). Equations (2-3) through (2-7) are adaptations of work performed by Thrun and Smeija [17].

$$\text{Output Error Generation} \quad e_i^k = d_i^k - y_i^k \quad (2-3)$$

$$\text{Hidden Layer Error} \quad e_j^{k-1} = \sum_{i=1}^{N_k} w_{ij}^k \sigma'(u_i^k) e_i^k \quad (2-4)$$

$$\text{Delta Weights (Output Layer)} \quad \Delta w_{ij}^k = \mu \sigma'(u_i^k) e_i^k y_j^{k-1} \quad (2-5)$$

$$\text{Delta Weights (Hidden Layer)} \quad \Delta w_{ij}^{k-1} = \mu \sigma'(u_i^{k-1}) e_i^{k-1} y_j^{k-2} \quad \text{or} \quad (2-6)$$

$$\Delta w_{ij}^{k-1} = \mu \sigma'(u_i^{k-1}) \left(\sum_{i=1}^{N_k} w_{ij}^k \sigma'(u_i^k) e_i^k \right) y_j^{k-2}$$

$$\text{Weight Update} \quad w_{new}^k = w_{old}^k + \Delta w_{ij}^k \quad (2-7)$$

Another excellent source of reference material can be found in the paper written by Rumelhart, Hinton and Williams [18].

Learning requires that first, retrieval take place, second, output activations be generated and, third, error terms be computed. Equation (2-3) is the output error computed between the desired signal and the output activation. This error is then backpropagated through the output layer PEs and output layer weights to obtain the hidden layer error. See equation (2-4). This is required for all networks containing more than one layer of weights (i.e., networks with hidden weight layers). Depending on the number of hidden layers in the network, the error may have to be further propagated through weight layers to obtain the error terms at *all hidden layer PEs*. Note in equation (2-4), N_k is the number of PEs in a given k layer. This is the same notation as shown in Figure 2-3.

Once all the PE layer errors have been obtained, the delta weight values for each layer are computed. Equation (2-5) illustrates the delta weight computation for the output layer weights. This array is $N_k \times N_{k-1}$ in dimension. The computation is composed of backing the error through the output layer PEs ($\sigma'(u_i^k) e_i^k$) and multiplying it by the learning rate, μ , and the previous layers' activation, y_j . Similarly, delta weight values can be computed for all hidden layer weights. See equation (2-6). The learning rate or step size, μ , found in (2-5) and (2-6) is used to control the rate of convergence in the weight adaptation.

Upon computing all the delta weight values for each layer of weights, the weight updating process begins. This consists of reading in the old weight (used in retrieval and error computations), adding the associated delta value and storing the new weight back to memory. This is shown in equation (2-7). This is also an $N_k \times N_{k-1}$ operation (per layer of weights).

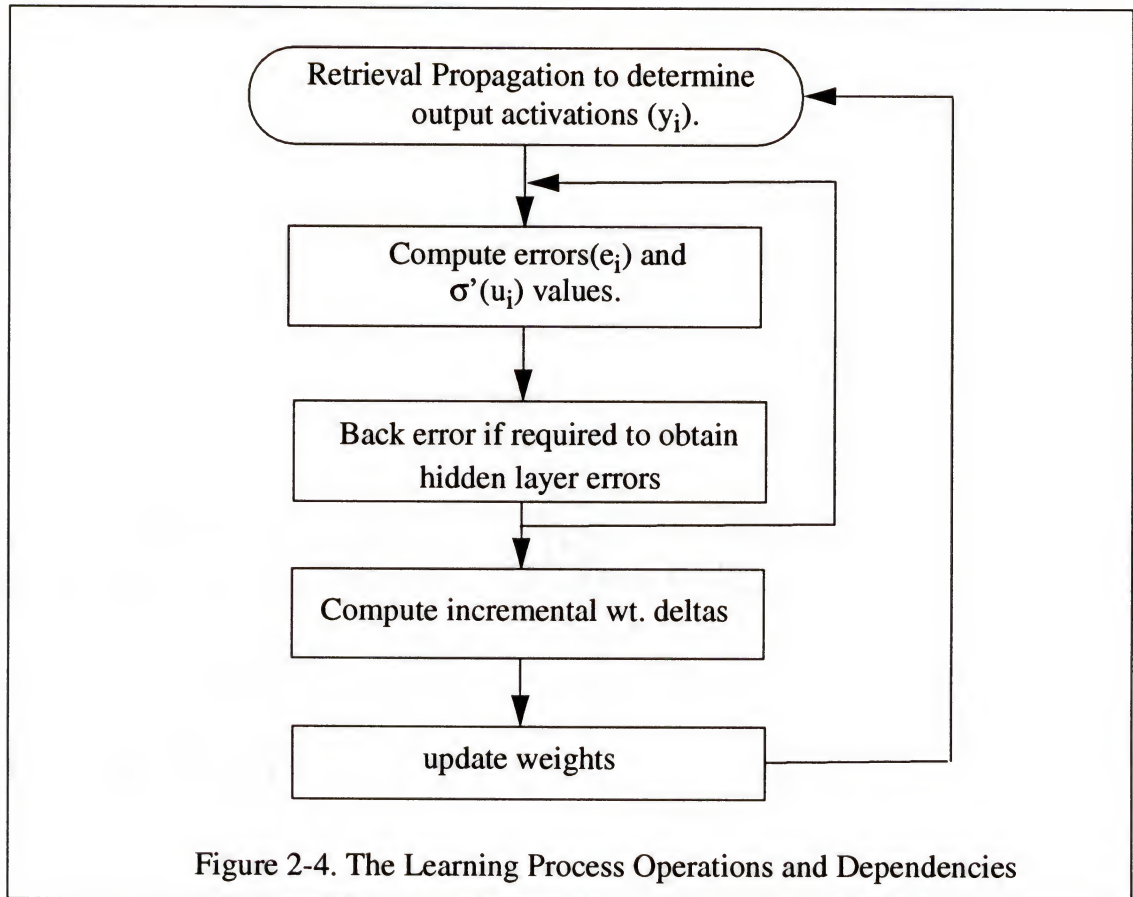
2.5 Parallelism and Pipeline Ability in the Backpropagation Algorithm

As in the retrieval mode equations, the hidden layer error vector is shown in summation form and not as the usual matrix notation. This is done both to exemplify the specific

multiply/accumulate requirements of the neural net algorithm and to solidify the network nomenclature that will be used throughout this work. By writing these equations as summations, it will be easier to observe the parallelism/pipeline ability in the algorithm, which is required for developing specific architectures later.

Backpropagation, like retrieval, contains a great deal of parallelism. The output error computation (2-3) can be performed in parallel by all output PEs. Similarly, the nonlinear transform values, $\sigma'(u_i)$, can be generated in parallel. A common nonlinear transform used in retrieval is the sigmoid function which is $(1+e^{-u})^{-1}$. When the derivative of this function is taken, the result yields $y_i(1-y_i)$; the activation (computed during retrieval) is multiplied by one minus the activation. Thus the ' u_i ' nets do not need to be saved from retrieval, only the resultant activations. The $y_i(1-y_i)$ quantities will be referred to as the derivative of the nonlinear transform function values and are found to be N_{k-1} in dimension. After these values are obtained, the backpropagation multiply/accumulate operations (2-4) can begin. Backpropagating the errors through a particular layer is very similar to computing a net quantity in the retrieval mode. If concurrent reading of the weight values is permitted, this can also proceed in parallel by each hidden layer PE. This, however, requires a mapping of one processing element (from a hardware point of view, i.e., DSP chip, neural IC, etc.) per PE in the layer and that the implementing element be able to perform operations (2-3) and (2-4). While this parallelism exists it poses many communication problems. For example, equation (2-4) also requires that each hidden layer PE be able to see the errors generated at the output layer. This again requires some type of concurrent read mechanism as in the reading of the weights.

Computing the delta weight values (2-5) can also be performed in parallel. This however, is an $N_k \times N_{k-1}$ operation, and previously we have mapped a single PE to a single



processor. Hence, the number of processors will have to be expanded such that each weight maps to a single physical processing unit. Under this mapping one processor computes one ij element in the Δw_{ij}^k array; the total number of processors, therefore, equals the total number of weights in the network. Each processor can then also update its assigned weight such that the group operates in parallel. While this is a possible solution, it becomes a very expensive solution when considering networks with hundreds or thousands of weights.

Pipelining network layers (horizontal parallelism) cannot be applied to backpropagation due to feedback nature of the process flow. See Figure 2-4. The weights cannot be updated until all errors (output and hidden) are obtained. Similarly, a new input vector cannot be presented to the net until all updating of the old weights has occurred.

In summary, during learning there is parallelism in a particular layer but pipelining between layers is extremely difficult. To exploit the parallelism found in a given layer, several problems must be overcome. During retrieval, all processing units were required to access the inputs for activation computation at the first layer of PEs. Similarly, during backpropagation the hidden layer PEs required all output errors for propagation in the reverse direction. Another problem is that the weights used during retrieval by a PE are not those used by the same PE computing the backpropagation of the error. Thus if physical processors are assigned to PEs they will need to be able to read different weight sets for each direction. This requires duplication of weights if they are to be stored locally in the processors or some global bank of weights that can be simultaneously accessed by all physical processors. Updating the weights is also a problem because the global bank needs to be simultaneously accessed again by several processors. Another mapping scheme was to assign a weight to an individual processor. However, this also requires an extensive inter-processor communication and is cost prohibitive. In closing, the purpose of this section was to point out the areas of parallelism in the learning algorithm and the problems associated with physically realizing this parallelism and, I hope, to spark the reader into thinking about possible solutions to the communication/weight storage problems.

CHAPTER 3 IMPLEMENTATIONS IN ARTIFICIAL NEURAL NET SYSTEMS

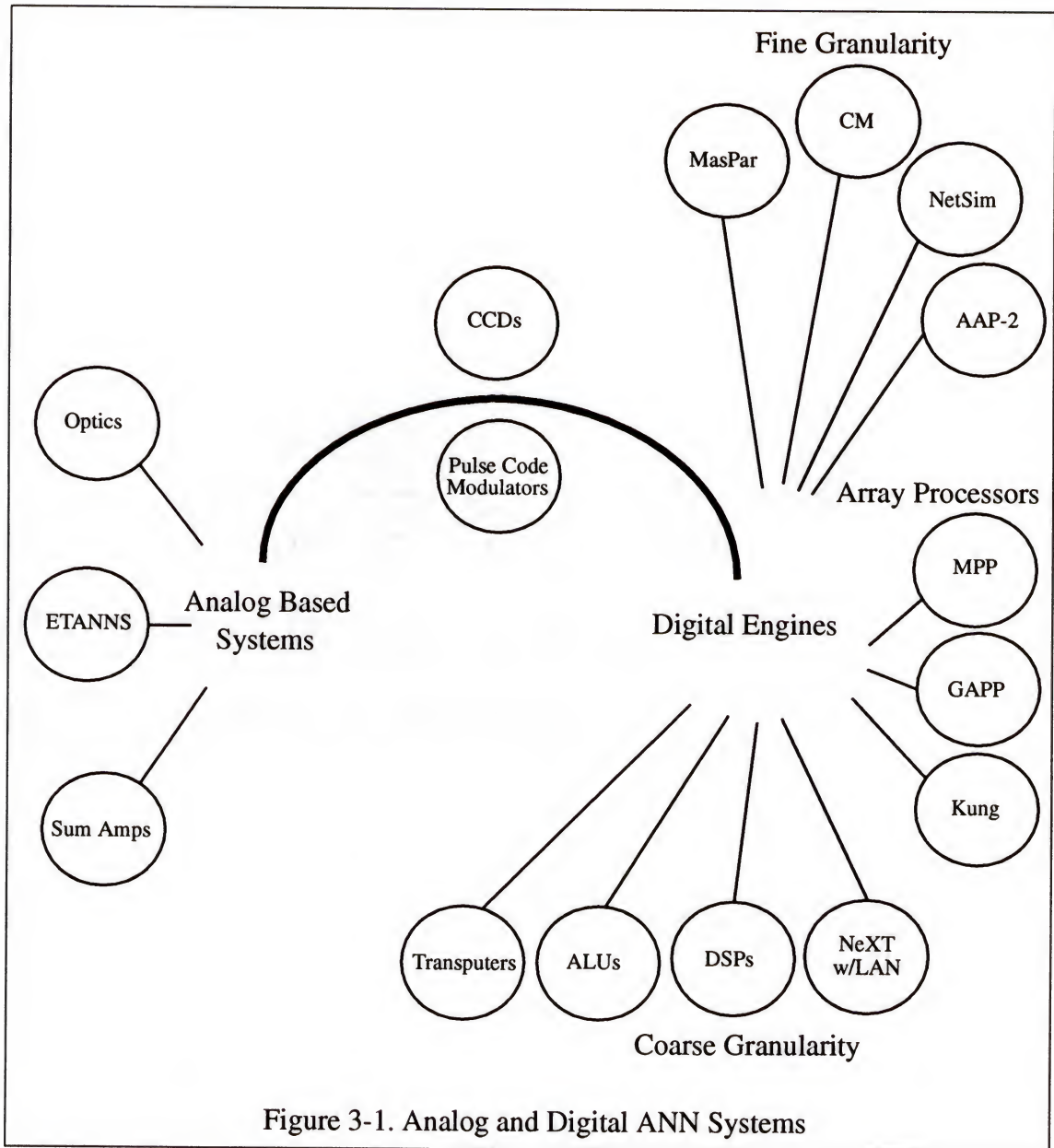
3.1 Introduction

Figure 3-1 illustrates the wide spectrum of systems that have been created or theoretically proposed to perform artificial neural network (ANN) computations in both the academic and commercial worlds. As shown, these can be categorized as being analog, digital, or a mixture of both. They can also be further grouped according to granularity. This chapter is dedicated to introducing the different approaches for speeding artificial neural processing and selects two interesting methods for extensive analysis.

3.2 Analog Domain

Given the analog systems shown in Figure 3-1, there appear to be three main divisions. However, all share the same main goal, ultrafast neural computation. Although optics appears to present the fastest connection rates, very few systems have been realized in this technology. The expenses of building such a system and required mechanical accuracies have made it practical only when very small networks and intense speeds are required [9] [19] [20].

Electrically trainable artificial neural networks (ETANNs) are one of the first NN hardware entries into the commercial sector. The researchers at INTEL Corporation have developed a chip based on an NMOS process that contains 62 neurons and 10,240 weights. The synapses are stored via a “floating gate” nonvolatile memory technology, which was originally proposed for high density memory chips [5]. The sum of products calculation is performed by first creating a differential current proportional to the input/synapse product and then feeding these products (currents) to a summing amplifier. While these chips are



very fast in the retrieval and learning mode, providing you use backpropagation, they have low synaptic resolution. The weights are said to have only 5 bits of accuracy [5] [21].

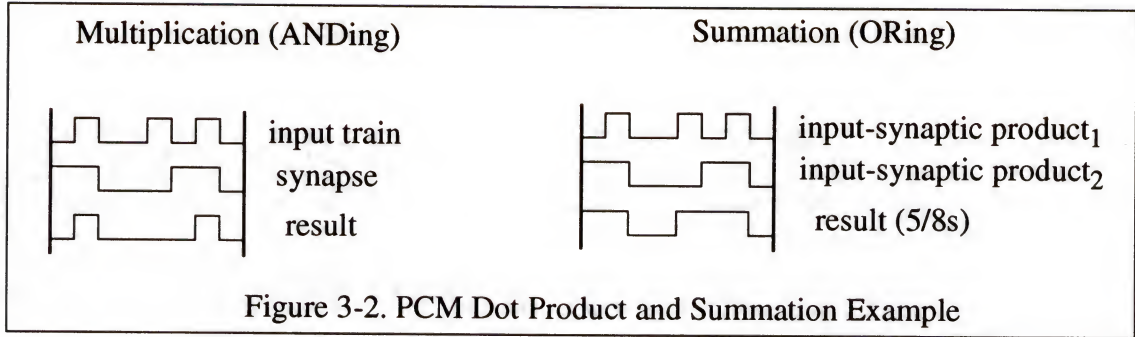
One last proposed implementation that strictly resides in the analog domain is a network topology based on simple summing amplifiers. Here the synaptic weights are represented by resistor values and are fed as inputs into a summing amplifier [22]. The nonlinear

squashing function is created by a second amplifier stage, which acts as a simple comparator. An actual system has been created using Op amps and programmable resistors [23]. Again, the weight resolution is low (5 bits), and noise (cross talk between connections) also raises some concerns. However, the system was used to control adaptively 21 actuators which are used to focus a celestial image through several lenses. The atmospheric error correction was performed real time, and the authors summarize that this would require a very fast and expensive mid-size computer to perform this same task digitally.

3.3 A Mixture of Both Worlds

Charge coupled devices (CCDs) and pulse code modulator-based architectures bridge the gap between the analog and digital domains. In CCDs, the weights are stored as an array of stored charges. These are then clocked through analog switches to analog accumulators (registers) where the summing occurs. The summing is performed similarly to switched capacitor designs where the capacitor (fabricated using CMOS technologies) is electrically connected (via transistor switches) to pass its charge to a storing device in a pre-defined short interval of time. Hence, the processing period is discrete but the amplitudes are not. Neural net literature on this subject is somewhat scarce and ill defined. However two papers written by Agranat and coworkers discuss a CCD realization [24] [25]. The CCD systems claim extremely high connection rates (10^9 connections per second), however, they do not mention the accuracy of the weights or output signals.

In a pulse code modulation realization, both the weights and inputs are represented as stochastic pulse trains. Here, activations are represented by firing frequencies and thus the synapse is used to modify this frequency [26]. For example, the required ANN dot product multiplication function consists of ANDing the appropriate input and synapse channels over a specified period of time (8 clock cycles). See Figure 3-2. In this example the input is equal to $3/8$ and the synapse is $1/2$ so the resulting product should be $3/16$. In this 8 clocks



per period of time the result is $2/8$ or $1/4$, which is therefore an approximation. However, greatly shrinking the clock period allows many more pulses to fit in a given window. This in turn greatly increases the accuracy of the dot product (multiplication).

Similarly, the summation portion of the dot product is performed by ORing the resulting input-synapse pulse trains originally created by the dot product. Again accuracy will greatly improve as the number of pulses per window increases. Included also in this summation process is a simple nonlinear squashing function. This occurs due to saturation which results when an entire period is filled with positive pulses (8 pulses for the example period shown in Figure 3-2). Thus, the actual numerical sum may be greater than the number of pulses in a window, it will always saturate to the maximum pulse per window value.

Although the pulse trains are discrete in both time and amplitude, the notion of representing an activation strength by *firing frequency* sets it apart from systems found in the digital domain. This type of processing is also a closer approximation of how the human brain's neurons pass information [27].

3.4 Digital Realizations

By far the greatest amount and widest variety of research reside in the digital domain. Digital realizations offer excellent synaptic resolution and minimize noise compared to their analog counterparts. In addition to this, it is much easier to expand with a digital

system. For example, if your network topology doubles, in an analog implementation the hardware must be doubled or time shared (providing this is possible). On a digital system, however, the hardware does not have to expand; the problem can be repartitioned on the existing system.

Figure 3-1 illustrates several areas of digitally related research and breaks them into three main classes, the first being fine granularity, the second, array processors and the third, coarse granularity. Fine granularity is characterized as having several hundred or even thousand processors where each PE has its own small chunk of local memory (e.g., 16K bytes). These systems are normally all single instruction multiple data (SIMD) machines. Examples are the Massively Parallel Processor (MasPar) [4], the Connection Machine (CM) [3], the Network Simulator (NetSim) [27] and the Massively Parallel Cellular Array Processor (AAP-2) [28].

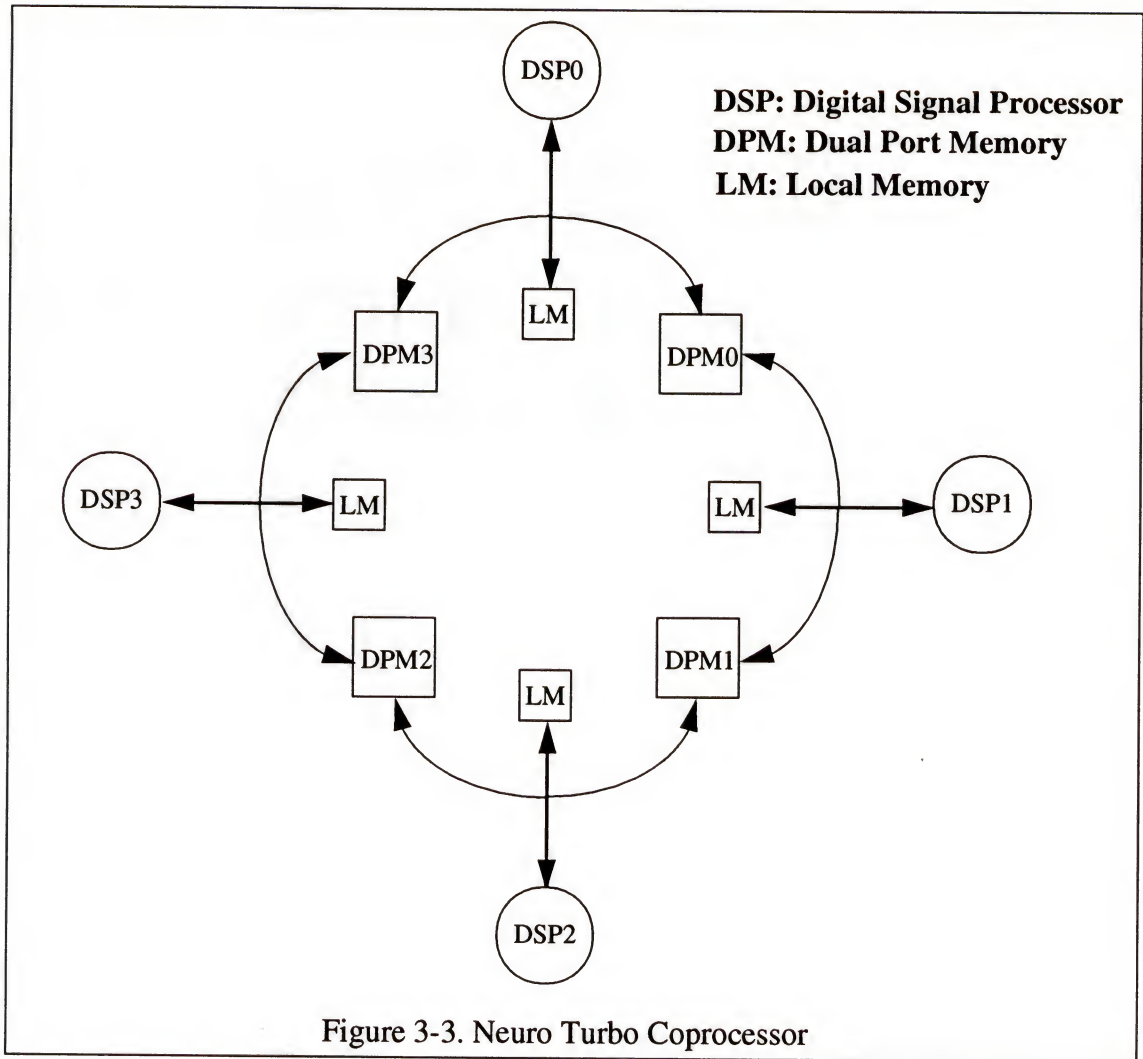
Array processors are shown as being in between the fine and coarse granularity classes because systems can be found in both regions. Under either class, array processors are composed of a group of special function processors that are connected usually in a linear, ring, or mesh array. Special function refers to the notion that the PE in the array usually only has a few specialized tasks, e.g., product, sum and nonlinear transformation, that are performed very rapidly. While these systems do have many similarities to those given in the fine granularity case, they differ in that the PE is less autonomous. In the array processor there is some type of centralized control. This control is used to produce rhythmic pulsed computations in which the array is then referred to as being systolic in nature. Examples of array-oriented backend ANN processors are the Massively Parallel Processor (MPP) [29], the Geometric Arithmetic Parallel Processor (GAPP) [30], and those proposed by S.Y. Kung [31] [32].

One last group of digital neural hardware, categorized as being of coarse granularity, are systems based on complex ALUs, transputers or digital signal processors (DSPs) [33][34][6]. Another example is a cluster of workstations running under a local area network environment. The DSP and transputer realizations are characterized as being loosely coupled and very autonomous. The complex ALU, however, is somewhat similar to an array-based system in that it has centralized control. The final example, workstations coupled via a LAN, offers an inexpensive parallel solution for research environments that already have workstations. This is referred to as computer clustering.

3.5 Neuro Turbo: a DSP-based Approach to Neural Computing

An example will now be discussed that illustrates how a multi-layer perceptron can be partitioned onto a given architecture. This is a DSP realization from the Nagoya Institute of Technology in Japan [6] called Neuro Turbo. The Neuro Turbo accelerator is a four processor, shared memory, ring architecture. See Figure 3-3. The processors are 24 bit, floating point DSPs. The shared memory or dual port memory found on both sides of a processor in the ring is 2K words in size. The processors also have their own working or local memory, which is a 64K word block. The activations and weights for a given network are stored locally amongst the four working memories, while all intermediate computations are stored in the dual port memory.

Because the weights are distributed evenly across the four processors, partial sums during either retrieval or backpropagation will have to be globally transmitted between processors. This is due to the fact that different weights are required for retrieval compared to those needed for backpropagation of error terms. Thus, if the weights required for retrieval are local to each processor (a layer of activation calculations being evenly divided across the four processors), partial sums will need to be passed during backpropagation of the error phase. Conversely, if the weights required for error backpropagation are made local,



partial sums (portions of the net term) will need to be globally transmitted between processors for retrieval. See Figure 3-4 for an example of weights stored in backpropagation alignment.

The question now becomes which of the two partitioning schemes to implement. To answer this, we must look again at the delta weight equations (2-5) and (2-6). Updating a weight requires the input value or previous layer activation (depending on which layer of weights we are updating in the net) at the originating point of the weight. In the case where weights have been stored locally for retrieval this means that each system processor must

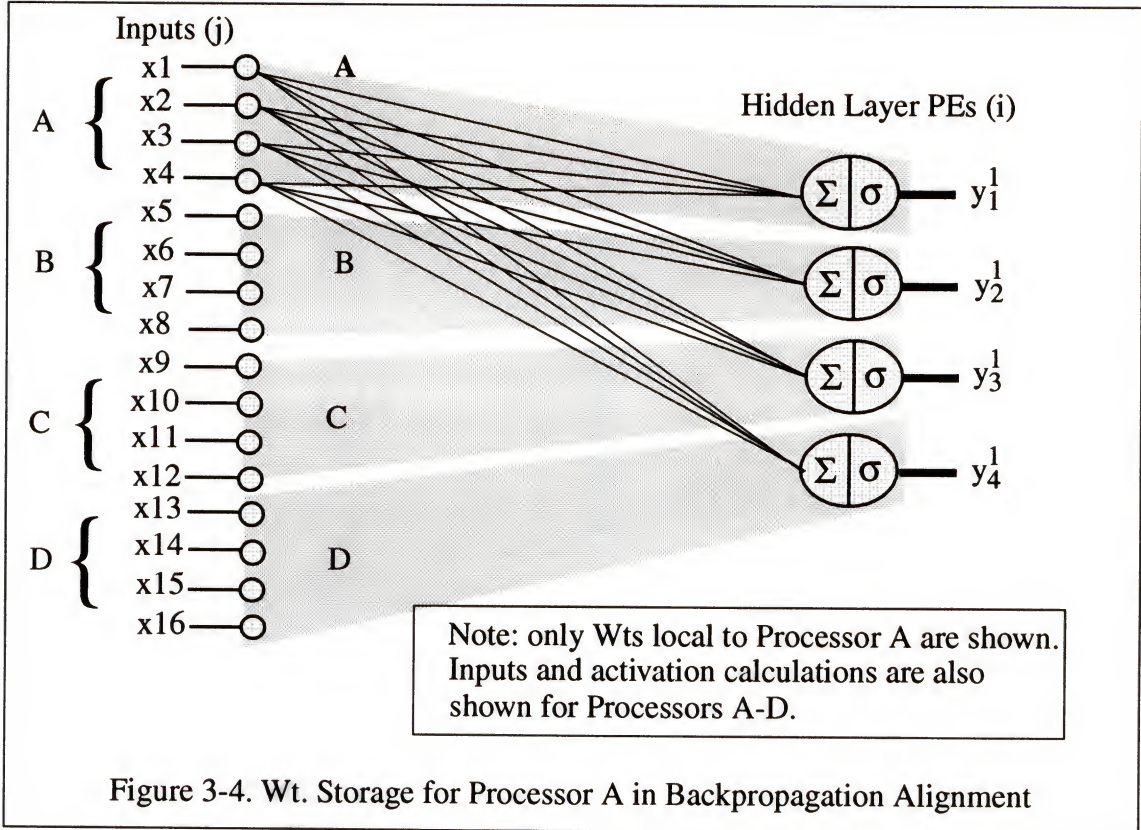
also contain all the previous layer inputs (activations). Comparing this to the case where the weights have been stored locally for backpropagation, the number of inputs required locally becomes $1/P$ where P is the number of processors in the system. See Figure 3-4. The weights for only processor A are shown ($P=4$) and the inputs required for updating the weights local to A are x_1 - x_4 . Thus partitioning according to backpropagation alignment is more efficient than partitioning according to forward propagation.

It should also be mentioned that one additional possibility is to duplicate the weights such that all the weights needed for computing assigned nets and error backpropagation are stored locally. Essential the number of weights stored locally are doubled. This eliminates the need for sending partial sums during retrieval and backpropagation but requires that both sets of weights be updated. Because the weights are an N^2 quantity (discussed in section 1.2), this is very undesirable for large networks. This additional processing requirement will be far more time consuming than globally transmitting partial sums as in the other two cases. The transmission of partial sums is an N size quantity.

3.5.1 Neuro Turbo's Partitioning According to Backpropagation Alignment

Partitioning according to backpropagation alignment was first published by Neuro Turbo researchers [6]. It turns out to be very efficient when local processing is compared with global communication. Because of this efficiency, the author adopted and extended this type of partitioning on a system composed of four Motorola DSP 56000s, and the results of this are presented in Chapter 8.

In addition, this scheme was also applied to a parallel cluster of NeXT workstations and a hybrid DSP system (CNEL Machine) being developed in our lab. Thus because of the value and usage of this partitioning scheme, it is presented in great detail. It is first described from a high level point of view and then in detailed equations (sections 3.5.2 and 3.5.3).



The partitioning first consists of dividing all the PEs in a layer across all the system processors and then the weights according to backpropagation assignment. See Figure 3-4. In this example the system processor count is $P=4$, the input layer N_0 is 16, and the hidden layer N_1 is 4. Individual processors are labeled A, B, C and D. Processor A is therefore shown to contain all weights corresponding to $j=1,2,3,4$ and $i=1,2,3,4$. Processor B similarly has weights $j=5,6,7,8$ and $i=1,2,3,4$ in local memory; processor C has all weights corresponding to $j=9,10,11,12$; and D has all weights with $j=13,14,15,16$.

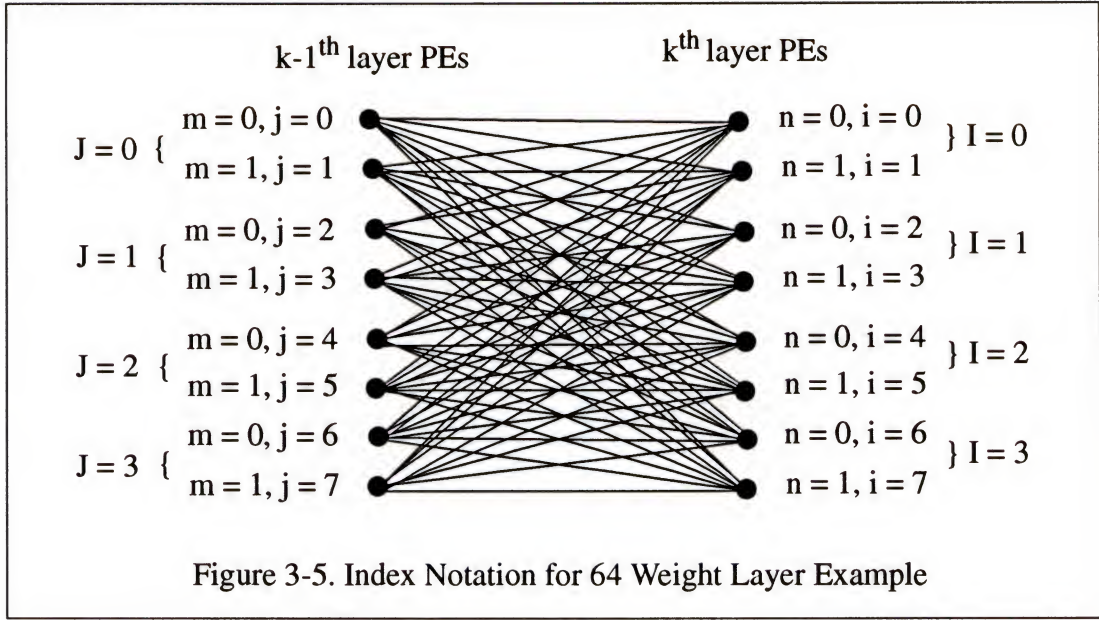
As mentioned earlier, because only the weights required for backpropagation are local to a processor, partial sums will need to be passed by processors A-D for retrieval. In Figure 3-4, processor A will compute partial sums (net terms) for all PEs in the hidden layer. Similarly, all other processors will generate net terms for all hidden layer PEs. These

partial sums will then be globally communicated between processors for generation of the final net value. The nonlinear transform will then be performed locally in parallel by each processor.

During backpropagation, output error terms (first generated locally) must be passed globally to all other processors. This is due to the backpropagation of error process requiring all output errors be local. Consult equation (2-4) and refer again to Figure 3-4. Assume for a moment that the hidden layer in Figure 3-4 is actually an output layer. Each processor will first compute a single local error term and then send this value to all other processors in the system. Having received all the errors, processors A-D can locally propagate the error through the PE and back it through its assigned weights. Thus backpropagation of the error through the weights can be performed concurrently by each system processor without any additional overhead (required weights are local). Similarly, the weights can also be updated in parallel (locally) by all processors.

3.5.2 Retrieval Equations

Specifically retrieval mode is performed in the following manner. The network is broken up so that for a given k layer, an individual 'i' PE's dot-product is computed across the 4 DSPs. This is best explained with another example. Suppose we are given a k^{th} layer of PEs where $N_k = 8$ and a $k-1^{\text{th}}$ layer of PEs where $N_{k-1} = 8$. See Figure 3-5. In this example the network is fully connected and so there are 64 total weights. The indices are first computed ahead of time as required to partition the network. I and J are the PE layer indices which correspond to each processor where $I = J = 0, 1, 2, \dots, P-1$ (recalling that P is the number of processors in the system); m and n are the individual PE indices that correspond to a PE assigned to a particular processor. Refer to Figure 3-5 in which $m = 0, 1, \dots, (N_{k-1}/4)-1$



and $n = 0, 1, \dots, (N_k/4) - 1$. These indices can be related to the i and j notation presented in Chapter 2 by equations (3-1) and (3-2).

$$\begin{array}{l} \text{Terminating PE} \\ \text{for a weight} \end{array} \quad i = \frac{N_k}{4} I + n \quad (3-1)$$

$$\begin{array}{l} \text{Originating PE} \\ \text{for a weight} \end{array} \quad j = \frac{N_{k-1}}{4} J + m \quad (3-2)$$

After the indices are computed, the weights are initialized (randomly created) in memory on each processor. Retrieval can now proceed and is shown by (3-5) through (3-9). The partitioned net is given in (3-3) and renamed according to Neuro Turbo's convention in (3-4). In our 64 weights per layer example there are 8 nets comprising N_{In} . Each individual net computation is then divided between the four processors as shown by (3-6) through (3-9). Thus each Neuro Turbo DSP will compute 1/4 of the dot-product sum for a given i PE. These partial calculations will then be recombined in a ring manner using the

*Net
Computation*

$$u_{\frac{N_k}{4}I+n} = \sum_{J=0}^3 \sum_{m=0}^{\frac{N_{k-1}}{4}-1} w_{(\frac{N_k}{4}I+n, \frac{N_{k-1}}{4}J+m)} y_{(\frac{N_k-1}{4}J+m)} \quad (3-3)$$

Neuro Turbo Net Notation

$$N_{In} = u_{\frac{N_k}{4}I+n} \quad (3-4)$$

Partial Nets

$$K_{IJn} = \sum_{m=0}^{\frac{N_{k-1}}{4}-1} W_{IJnm} Y_{Jm} \quad (3-5)$$

Relation between N and Partial Net

$$N_{0n} = K_{00n} + K_{01n} + K_{02n} + K_{03n} \quad (3-6)$$

$$N_{1n} = K_{10n} + K_{11n} + K_{12n} + K_{13n} \quad (3-7)$$

$$N_{2n} = K_{20n} + K_{21n} + K_{22n} + K_{23n} \quad (3-8)$$

$$N_{3n} = K_{30n} + K_{31n} + K_{32n} + K_{33n} \quad (3-9)$$

dual port memory. Before discussing this, however, the storage of inputs, activations and weights needs to be presented and reviewed.

Figure 3-6 illustrates the k-1 layer activations, weights and partitioned net owners (DSPs). One immediate advantage shown by Figure 3-6 is that no duplication of data is present; the data are uniformly distributed among all four of the processors. It is also important to note that the inputs and weights are stored such that the computation in equation (3-5) can be performed local to a processor. This is performed in the following manner.

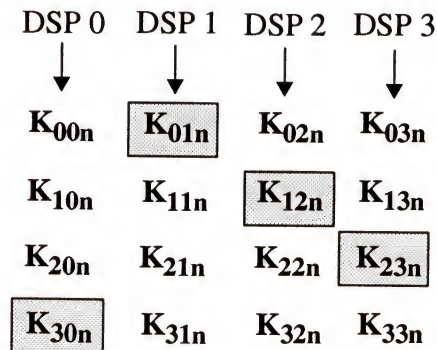
The K_{IJn} partitioned nets are owned by a given processor as shown in Figure 3-7. All processors will first begin by computing the shaded boxed partial net assigned to them in Figure 3-7. Upon completion, they will compute the next partial net group residing below

Working Memory Data Distribution

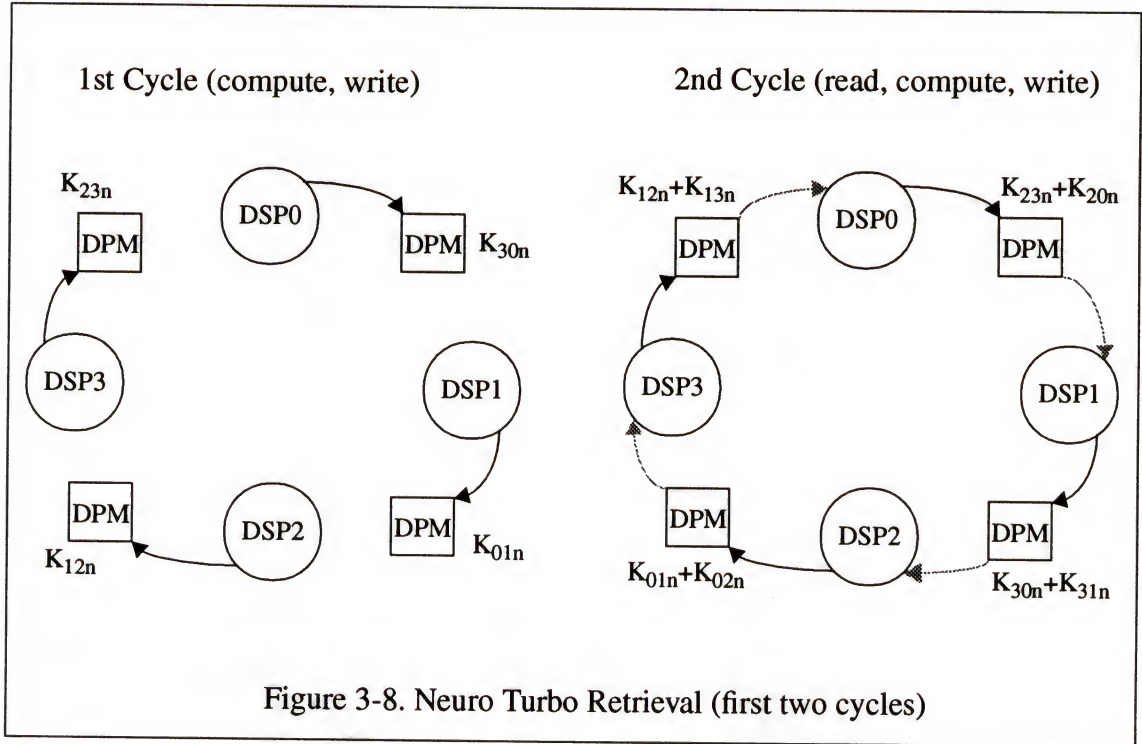
Layer	DSP 0	DSP 1	DSP 2	DSP 3
k-1	Y_{0m}	Y_{1m}	Y_{2m}	Y_{3m}
from k-1 layer to the k layer	W_{00nm}	W_{01nm}	W_{02nm}	W_{03nm}
	W_{10nm}	W_{11nm}	W_{12nm}	W_{13nm}
	W_{20nm}	W_{21nm}	W_{22nm}	W_{23nm}
	W_{30nm}	W_{31nm}	W_{32nm}	W_{33nm}

Figure 3-6. Weight, Activation Storage

DSPs compute the 'K' partitioned dot-product sums in their column.

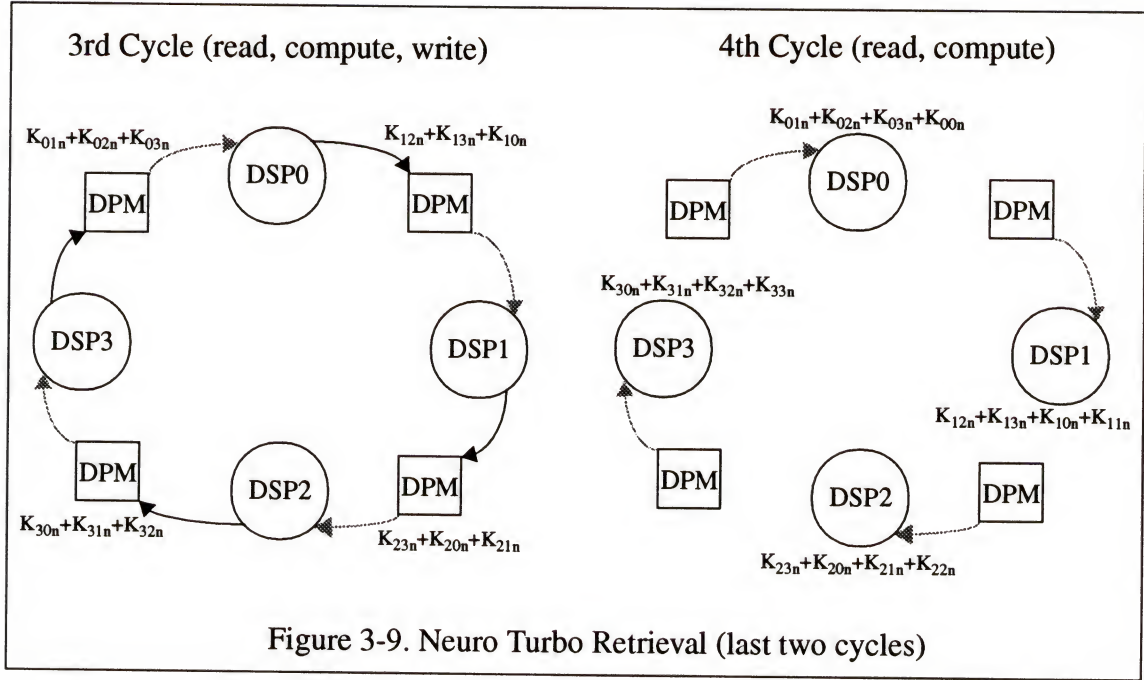
Figure 3-7. K_{II_n} Processing Ownership

the original shaded group of computations. If the original grayed partial net is at the bottom of the column (as in DSP 0), the next partial net is taken from the top of the column. This repeats until all four partial nets are computed by each processor. Intermediate results are passed through the dual port memories.



This can be specifically described as in the starting pass, DSP0 will compute K_{30n} , DSP1 will compute K_{01n} , DSP2 will compute K_{12n} , and DSP3 will compute K_{23n} . The results will then be *stored clockwise in the dual port memories*. See the left side of Figure 3-8. The write cycle is shown as a black arrow and the read is gray. After completing a cycle, results are stored in the dual port memories which is written next to the DPM in Figure 3-8.

The 'K' scheduling shown in the 1st cycle therefore follows a diagonal pattern in Figure 3-7. In the 2nd cycle, the DSPs will first read from the dual port memory that is counter clockwise from them, compute the new sum, and write results to the clockwise dual port memory. See the right side of Figure 3-9. In cycle 2, the DSPs compute as follows: DSP0 will compute K_{20n} , DSP1 will compute K_{31n} , DSP2 will compute K_{02n} , and DSP3 will compute K_{13n} . The 3rd and 4th cycles are shown in Figure 3-9. The 3rd cycle is similar



to the 2nd cycle; the previous computed sums are read counter clockwise, computed with the next 'K' partial net and stored in the clockwise direction. Cycle 4 is a read and compute only. Upon creating the N_{In} dot-products after the 4th cycle, the DSPs will perform the necessary nonlinear transform in parallel using a table look-up method. The Y_{Im} activations associated with a particular DSP are then stored in the corresponding working memory. This process can then be repeated to compute the next layer activations. The next layer weights and recently computed activations are stored as illustrated before in Figure 3-6.

3.5.3 Backpropagation of Error

Neuro Turbo backpropagation is computed in a similar manner as retrieval. The error is propagated from the output layer to the hidden layer using equations in (3-10)-(3-18). Note that Neuro Turbo researchers define their error vectors to be to the left of the PE and that the retrieval mode nonlinear transformation is a table look-up version of the sigmoid function. See (3-10). (3-11) is the hidden layer error generation. Similarly, the hidden layer

Error Definition $\delta_i = (d_i - y_i) (\sigma'(u_i)) = (d_i - y_i) (y_i (1 - y_i)) \quad (3-10)$

Hidden Layer Error $\delta_j = y_i (1 - y_i) \sum_{i=0}^{N_k-1} w_{ij} \delta_i \quad (3-11)$

error vector is taken at the input of the PEs instead of at the output. The hidden layer error equation (3-11) is then re-written in terms of Neuro Turbo's I,J,m and n indices to yield (3-12). The hidden layer error vector has the same notation as that shown by the PEs and weights in Figure 3-5. Employing the 64 wt. layer example as before, the eight hidden layer errors are grouped in m pairs and are spread across the four DSPs (J=0,1,2,3). The reason

Hidden Error (re-written)

$$\delta_{\frac{N_{k-1}}{4}J+m} = Y_{\frac{N_{k-1}}{4}J+m} \left(1 - Y_{\frac{N_{k-1}}{4}J+m} \right) \sum_{I=0}^3 \sum_{n=0}^{\frac{N_k}{4}-1} w_{(\frac{N_k}{4}I+n)(\frac{N_{k-1}}{4}J+m)} \delta_{\frac{N_k}{4}I+n} \quad (3-12)$$

(3-12) is shown is so that the reader can refer back to the hidden layer error in (2-4) and observe how the indices function for partitioning. The J index will again be used to partition these error computations across the four processors (J=0,1,2,3)

Equation (3-13) is a simplified convention for the output error backed through a PE used in (3-12). From (3-13) and Neuro Turbo's weight notation, the hidden layer error can

Neuro Turbo's Output Error $D_{In} = \delta_{\frac{N_k}{4}I+n} \quad (3-13)$

be more simply written as in (3-14). Note this error is defined as 'E' rather than 'δ' because it is taken at the *output of the hidden layer PE* not to the left of the PE. These error computations can now be partitioned across the four processors as in (3-15) through (3-18). Each E_{Jm} subdivision of hidden error computation will be executed completely by one DSP.

$$\text{Error Backed Through PE} \quad E_{Jm} = \sum_{n=0}^{\frac{N_k}{4}-1} W_{IJnm} D_{In} \quad (3-14)$$

Partitioned Hidden Layer Errors

$$D_{0m} = Y_{0m} (1 - Y_{0m}) (E_{00m} + E_{10m} + E_{20m} + E_{30m}) \quad (3-15)$$

$$D_{1m} = Y_{1m} (1 - Y_{1m}) (E_{01m} + E_{11m} + E_{21m} + E_{31m}) \quad (3-16)$$

$$D_{2m} = Y_{2m} (1 - Y_{2m}) (E_{02m} + E_{12m} + E_{22m} + E_{32m}) \quad (3-17)$$

$$D_{3m} = Y_{3m} (1 - Y_{3m}) (E_{03m} + E_{13m} + E_{23m} + E_{33m}) \quad (3-18)$$

The final cycle of retrieval (right hand side of Figure 3-6) ensures that the activations generated by each processor once again correspond to that processor. For example, the completed nets found in DSP0 all correspond to $I=0$, the sums in DSP1 are $I=1$ and so forth for all other processors. This allows the desired signal to be split evenly among all DSPs and the output error to be computed locally in parallel across all DSPs. This error can then be propagated through the associated PE in parallel across all DSPs also (activations are local). This generates the necessary D_{In} terms required by (3-14). Once computed, these values will be passed around the ring in a similar manner as the partial nets were during retrieval. However before this is shown, the local storage map will again be shown for clarity purposes. See Figure 3-10. This illustrates all the quantities local to a DSP.

Having computed the D_{In} terms, they are used to generate the hidden error E_{Jm} terms given by (3-14). See Figure 3-11. DSP0 will compute E_{0m} in (3-15), DSP1 will compute E_{1m} , etc... In the first cycle all DSPs will create their first partitioned error sum terms as required by equations (3-15)-(3-18). After this computation, the locally computed D_{In} errors are stored clockwise in the Dual Port Memory. In cycle 2, the DSPs first read from

Working Memory Data Distribution

Layer	DSP 0	DSP 1	DSP 2	DSP 3
k	Y_{0n}	Y_{1n}	Y_{2n}	Y_{3n}
k-1	Y_{0m}	Y_{1m}	Y_{2m}	Y_{3m}
from k-1 layer to the k layer	W_{00nm}	W_{01nm}	W_{02nm}	W_{03nm}
	W_{10nm}	W_{11nm}	W_{12nm}	W_{13nm}
	W_{20nm}	W_{21nm}	W_{22nm}	W_{23nm}
	W_{30nm}	W_{31nm}	W_{32nm}	W_{33nm}
k	D_{0n}	D_{1n}	D_{2n}	D_{3n}

Figure 3-10. Working Memory Contents after the Output Error Computations

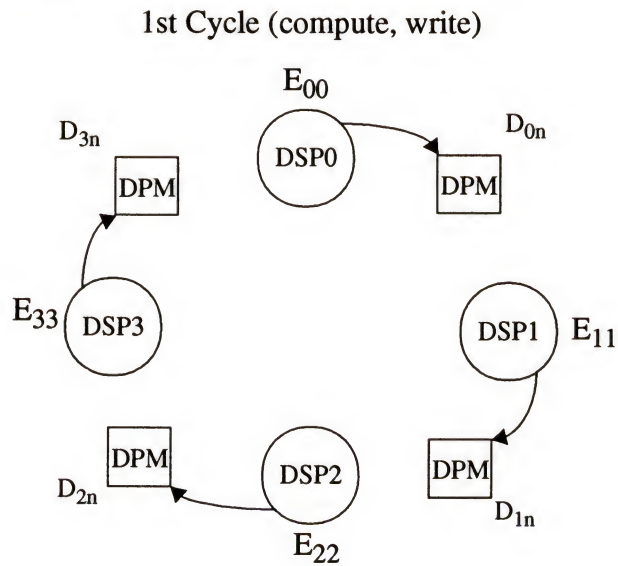
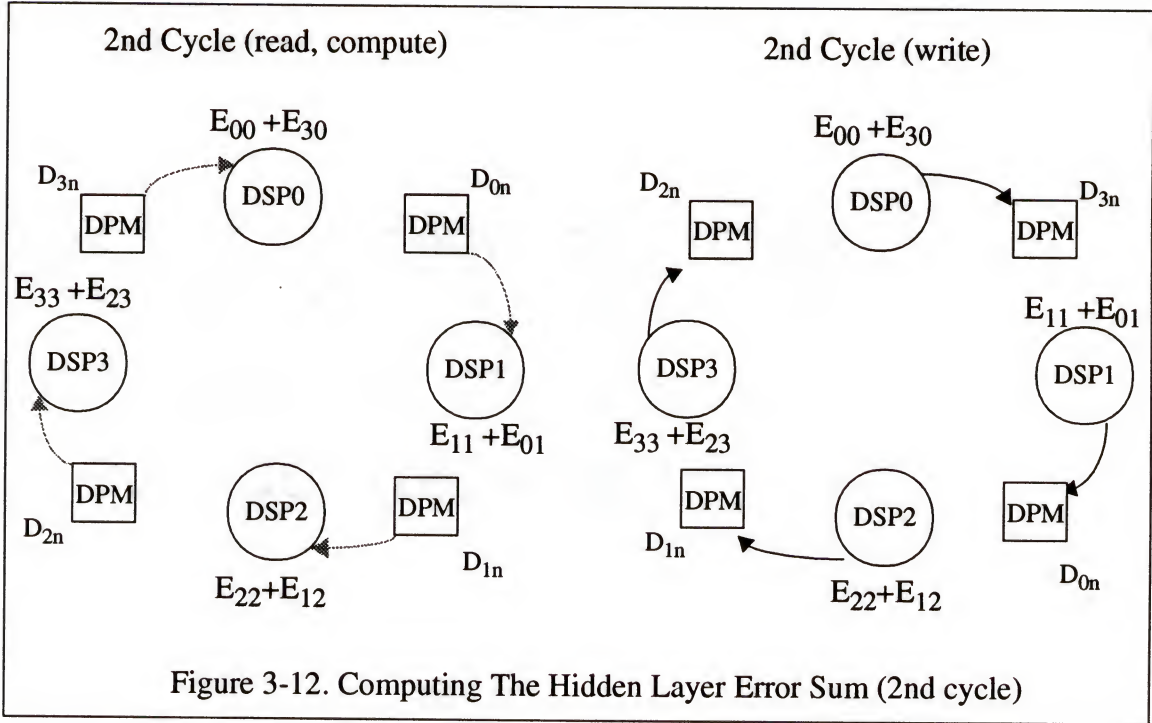


Figure 3-11. Computing The Hidden Layer Error Sum (1st cycle)



the counter clockwise DPM then compute the next error sum E_{Jm} term. Following this, D_{In} is then again sent clockwise. See Figure 3-12. This repeats until all E_{IJ} sums are computed in all four DSPs. The $k-1^{th}$ nonlinear derivative, $Y_{Jm}(1-Y_{Jm})$, can now be multiplied in parallel on each DSP and equations (3-15) through (3-18) can be completed.

At this point both the output and hidden layer vectors have been computed. If there is another hidden layer error calculation, the process will repeat with the recently computed D_{Jm} hidden error values becoming the new D_{In} terms in (3-14). for the next hidden layer error calculations.

3.5.4 Neuro Turbo Delta Weight Calculation

The weights shown in Figures 3-6 and 3-10 are updated according to equation (3-19). This is very similar to the update found in (2-5). In (3-19) the Y_{Jm} terms are local to the DSPs and are stored in the corresponding working memories; however, only one of the four D_{In}

(δ_i) terms is resident locally. Recalling that for DSP0, $J = 0$ and $I = 0, 1, 2, 3$, the corresponding output layer weights will require D_{0n} , D_{1n} , D_{2n} , and D_{3n} for their Δw_{ij} calculation.

Therefore one of two cases must occur; either the weights were updated immediately after the output error terms were passed in backpropagation (Figures 3-11 and 3-12) or the D_{In} terms are preserved in dual port memory (not written over) each time a backpropagation cycle occurs. In the last case all D_{In} terms would be local to the DSPs and the weight update could proceed with no further required communication.

$$\text{Delta Weight} \quad \Delta W_{IJ} = \mu D_{In} Y_{Jm} \quad (3-19)$$

In the case where the hidden layer weights are to be updated, if there is no error to be further propagated back (to another hidden layer), the D_{Jm} (D_{In}^{k-1}) terms will have to be passed circularly for weight updating. Unfortunately, the Neuro Turbo researchers only specify how the layer errors are computed but not how the actual weight updating proceeds. In any event once the D_{In} error terms are local to a DSP, the weight update will proceed as described by (3-19) and (2-7).

3.6 S. Y. Kung's Systolic Array Neural Network Implementations

One last example of partitioning a multi-layer perceptron onto a specific given architecture is now presented. This consists of the research performed by one of the most famous neural partitioning researchers today, Dr. S. Y. Kung, at Princeton University. Dr. Kung's interests have been primarily in the area of partitioning various digital signal processing algorithms onto array processor systems [31] [32]. Presented is a method by which multi-layer perceptrons can be partitioned onto a systolic array ring. Retrieval will first be covered in detail and then backpropagation will be briefly discussed.

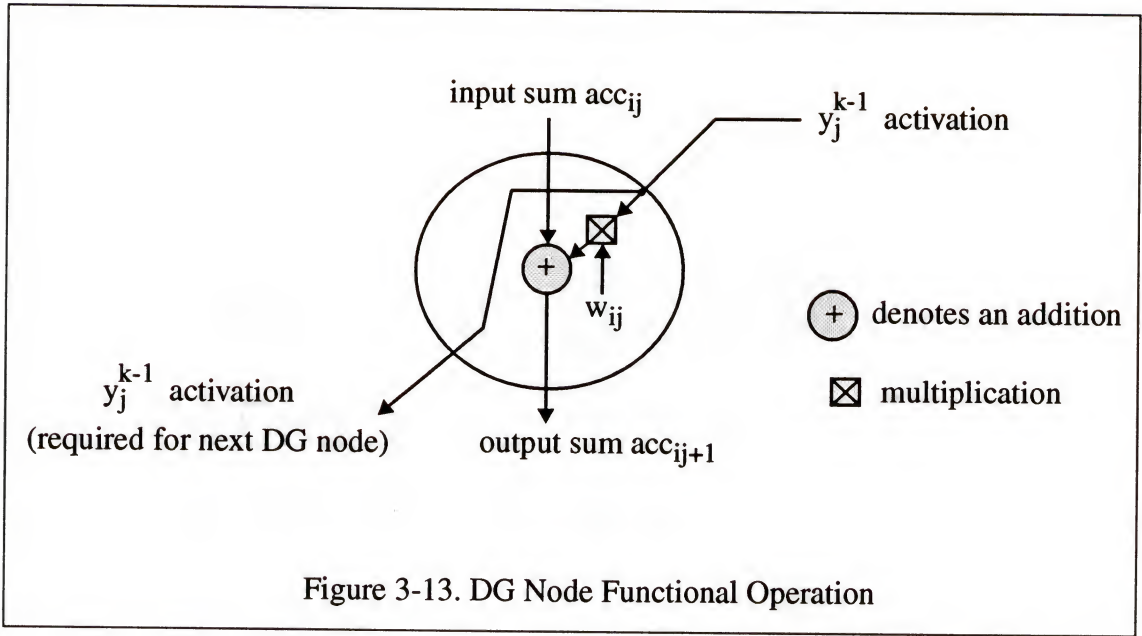
3.6.1 Retrieval

The method for partitioning any algorithm onto a systolic array first consists of determining the dependence graph (DG) for the algorithm's data flow. The resultant DG is then mapped to a corresponding array processor assignment and schedule assignment. Once the assignments have been made, a specified group of array processors can be selected that will compute under the guidelines provided by the schedule assignment.

The data dependence is comprised of nodes which represent computations and arcs which specify the data dependencies between computations. The purpose of the DG is to graphically express the recurrence and parallelism in a given algorithm. A DG will now be designed which maps onto it the retrieval mode.

Referring back to equation (2-1) where $y_i^k = \sigma \left(\sum_{j=1}^{N_k} w_{ij}^k y_j^{k-1} \right)$, the single layer

computation is formulated as a consecutive matrix-vector multiplication (MVM) problem interleaved with a nonlinear transformation. The MVM has a characteristic DG that has a computation node for every element (weight) in the matrix. This is due to the fact that every element in the matrix will undergo one computation. Another key rule is that the input data should be aligned in the same direction as the output data in the DG node. This also will apply to the computed sum from the past node and the generated output sum. This will become clearer after Figures 3-13 and 3-14 are viewed. These rules allow for cascading several copies of DGs to one another. See Figure 3-13 for the basic DG node that will be used in the MVM problem. It should be apparent in Figure 3-13 that the outputs follow the same direction as the inputs which meets the requirement for cascading several nodes and DGs comprised of these nodes. The variables acc_{ij} and acc_{ij+1} will correspond to the partial accumulations (net) that is required by equation (2-1). The summation in (2-1) is broken into $i \times N_k$ accumulations where the final net sum, $u_i = acc_{iN_k}$.



Combining several of these DG nodes (one per weight), a single layer retrieval DG can be derived. See Figure 3-14. The bottom layer corresponds to the nonlinearity which performs the squashing function of the final computed sums (u_i 's). Note in Figure 3-14 that because the inputs (top) and outputs (bottom) are aligned in the same direction, copies of this DG can be stacked (multiple network layers in retrieval mode). Another important item to observe is the wrap around effect in the DG. The inputs to the nodes wrap around in both the i and j direction. This will prove advantageous for implementing the projection in a systolic array ring.

The dashed lines in Figure 3-14 represent a scheduling plane where all the computations in the plane are performed concurrently. The scheduling plane is always normal to the scheduling vector, $\vec{s}$. Hence, the scheduling vector points in the j direction. There is also a projection vector, $\vec{d}$, which will also point in the j direction. This corresponds to a horizontal hyperplane which maps N_k nodes (e.g., $i = 1$ to N_k , $j = 1$ DG nodes) to N_k array processors. These vectors and associated hyperplanes are determined using the following

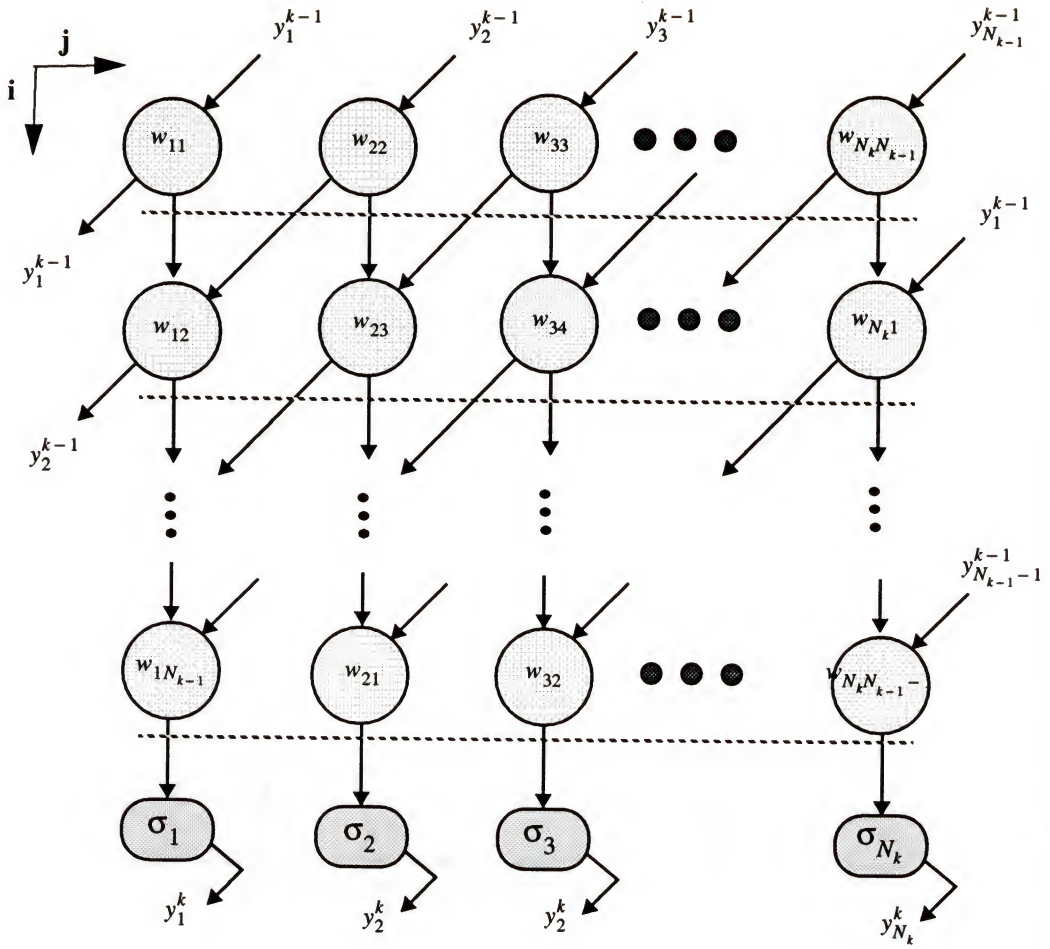


Figure 3-14. Single Layer Retrieval Mode Dependency Graph (DG)

conditions. The first is $\bar{s}^T \bar{e} > 0$; where $\bar{s}$ is the scheduling vector and $\bar{e}$ is any data dependence vector in the DG. Therefore in Figure 3-14, $\bar{s}$ cannot be in either the i or $i = j$ directions. A second condition is that $\bar{s}^T \bar{d} > 0$; which implies that s is never normal to $\bar{d}$ and the angle between the two vectors is less than 90 degrees. The resultant array processor projection derived from the DG in Figure 3-14 is shown in Figure 3-15.

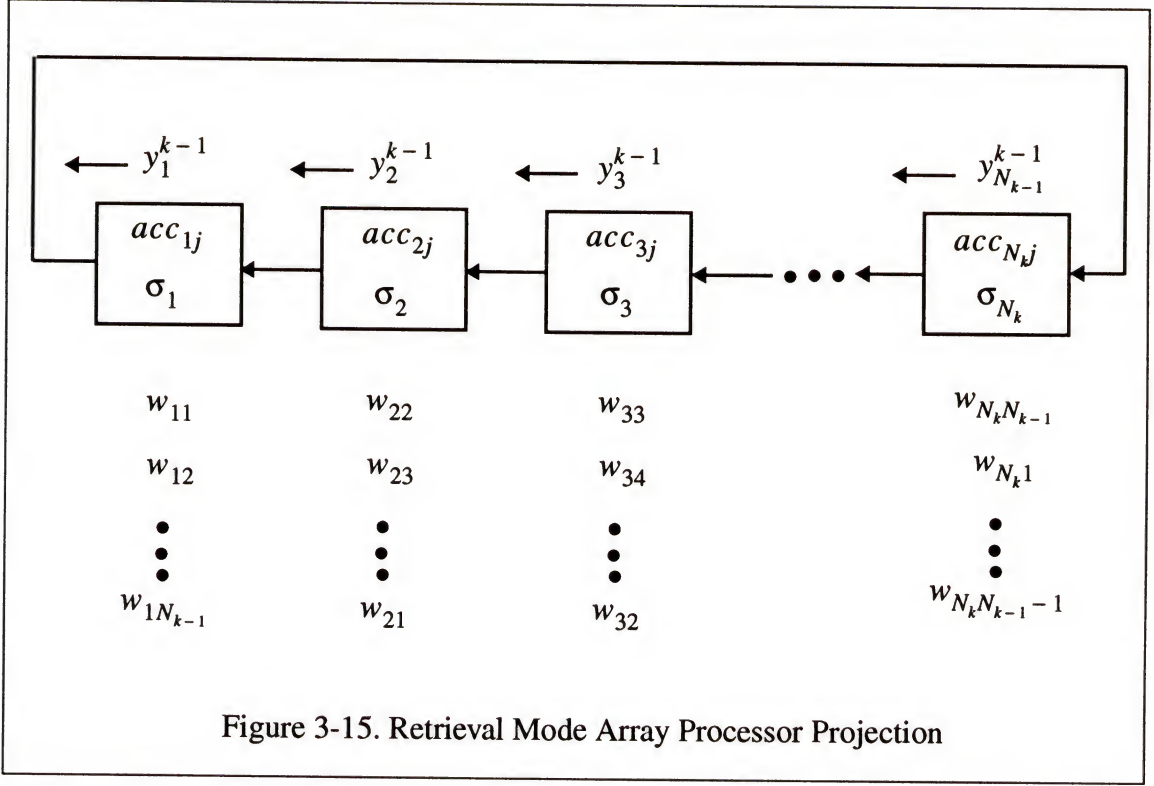
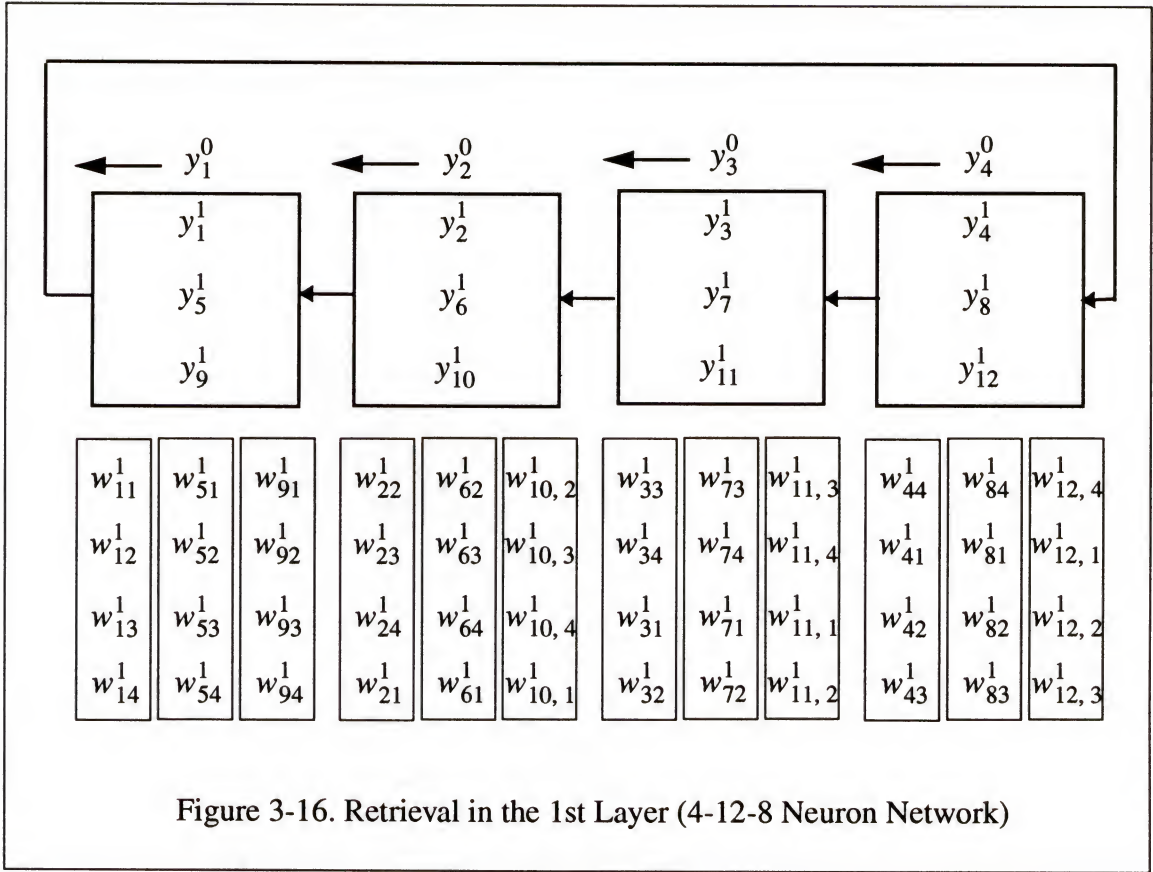


Figure 3-15 consists of a ring systolic architecture comprised of N_k processing units. A column of weights from the DG is stored in each processing unit. A cycle will consist of multiplying the current $k-1$ activation with one weight and adding this to the old sum (acc). After every cycle, the $k-1$ activations are passed clockwise to the next processing unit. N_{k-1} cycles are required to compute the N_k nets (in parallel). Once these nets have been obtained, they are all simultaneously passed through the nonlinear transform to produce the activations, $\sigma_1, \sigma_2, \sigma_3, \dots, \sigma_{N_k}$.

Thus far, the retrieval mode mapping for a particular layer in a network has been covered. However what about the more common case where a network consists of multiple layers of varying numbers of PEs? For example, suppose we are given a network with 4 input PEs, 12 hidden PEs, and 8 output PEs. For the 1st layer retrieval, 12 array processing

units are required and the input vector elements must be copied twice to produce the 3 inputs data sets spread across the 12 units. In order to compute the next layer activations, the number of array units must be shrunk to 8 to match the output PEs and the 12 inputs must somehow must spread across the 8 array units. This presents several logistic hardware problems such as how to switch units in and out of a ring and how to add dummy units for storing more inputs (activations from the previous layer) when they exceed the number of activations being computed for the next layer. To solve these problems, Dr. Kung suggests that *a ring array be created which corresponds to the smallest number of PEs in a given layer of the network* and that *all other layers be integer multiples of this number*. In the 4-12-8 PEs per layer example, the array is therefore chosen to be 4 units. If the layers are not exact multiples of the array, it is suggested that one should artificially pad a number of “no-operation” neural units to obtain the integer multiple. These dummy units do not compute any useful values but are used in timing the cycling of the inputs. Return to the 4-12-8 neuron case, Figure 3-16 illustrates the PE to array unit mapping and appropriate schedule. The 12 activations are computed one at a time in each array unit. The previous layer activations are shifted to the neighboring clockwise array unit after every multiplication by a weight. The weights are distributed and stored internally amongst the four array units. Hence, in order to compute y_1^1 , the inputs are shifted 4 times clockwise to compute the dot-product sum created with the first column of weights. Simultaneously, y_2^1 , y_3^1 , and y_4^1 are also computed in the other units. As before, when the inputs are shifted four times, it forms one complete cycle.

Therefore it takes approximately three cycles (12 shifts) to compute all the nets. Once the three nets have been computed in each array unit, the nonlinear transformation is serially generated to obtain the new three activations per array unit. Upon computing the k-1 layer of activations, the k layer retrieval commences. See Figure 3-17. The eight outer



layer activation calculations, y_i^2 , are distributed as pairs among the four array processors. The inputs are divided into three groups of four; a group is then shifted around the ring array as before. A complete shifting of data around (through) the ring is again deemed a cycle. While the data is shifting through the ring, weights are shifted from internal memory to perform the necessary dot-product sum. Note the ordering of the weights. Thus, three cycles (for the three groups of data) are required to compute the dot-product sums for y_1^2 through y_4^2 . Another three cycles will then be expended to create the remaining dot-products for the y_5^2 through y_8^2 activations. In summary, six cycles are required to compute all eight $k=2$ nets; these are then passed through the nonlinear transform to obtain the actual activation.

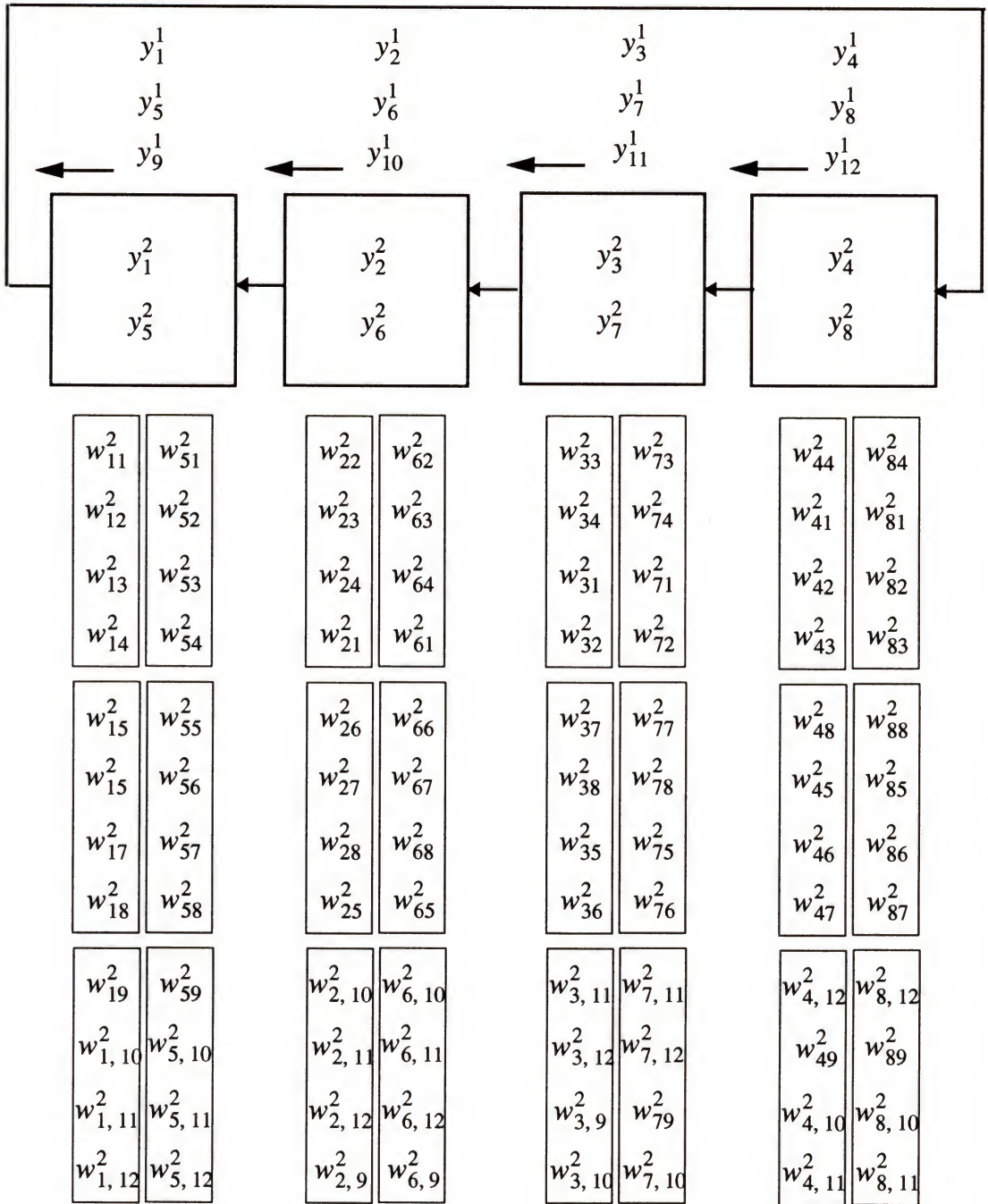


Figure 3-17. Retrieval in the 2nd Layer (4-12-8 Neuron Network)

3.6.2 Backpropagation

Backpropagation is broken into two separate processes, the first being a vector matrix multiply (VMM) which computes the hidden layer error and second an outer product updating (OPU). The OPU is the operation which updates the weights. In equation (2-4),

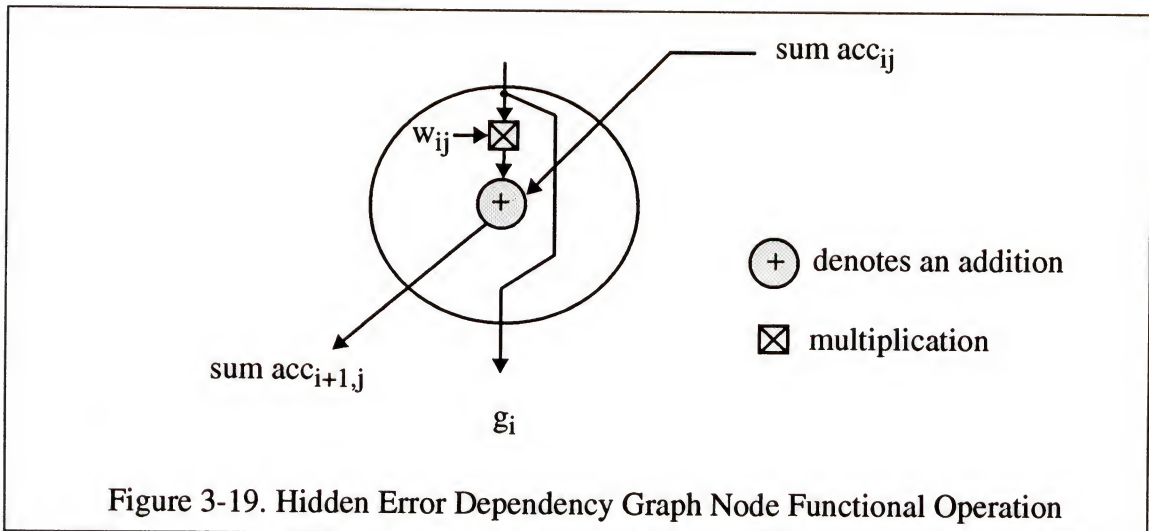
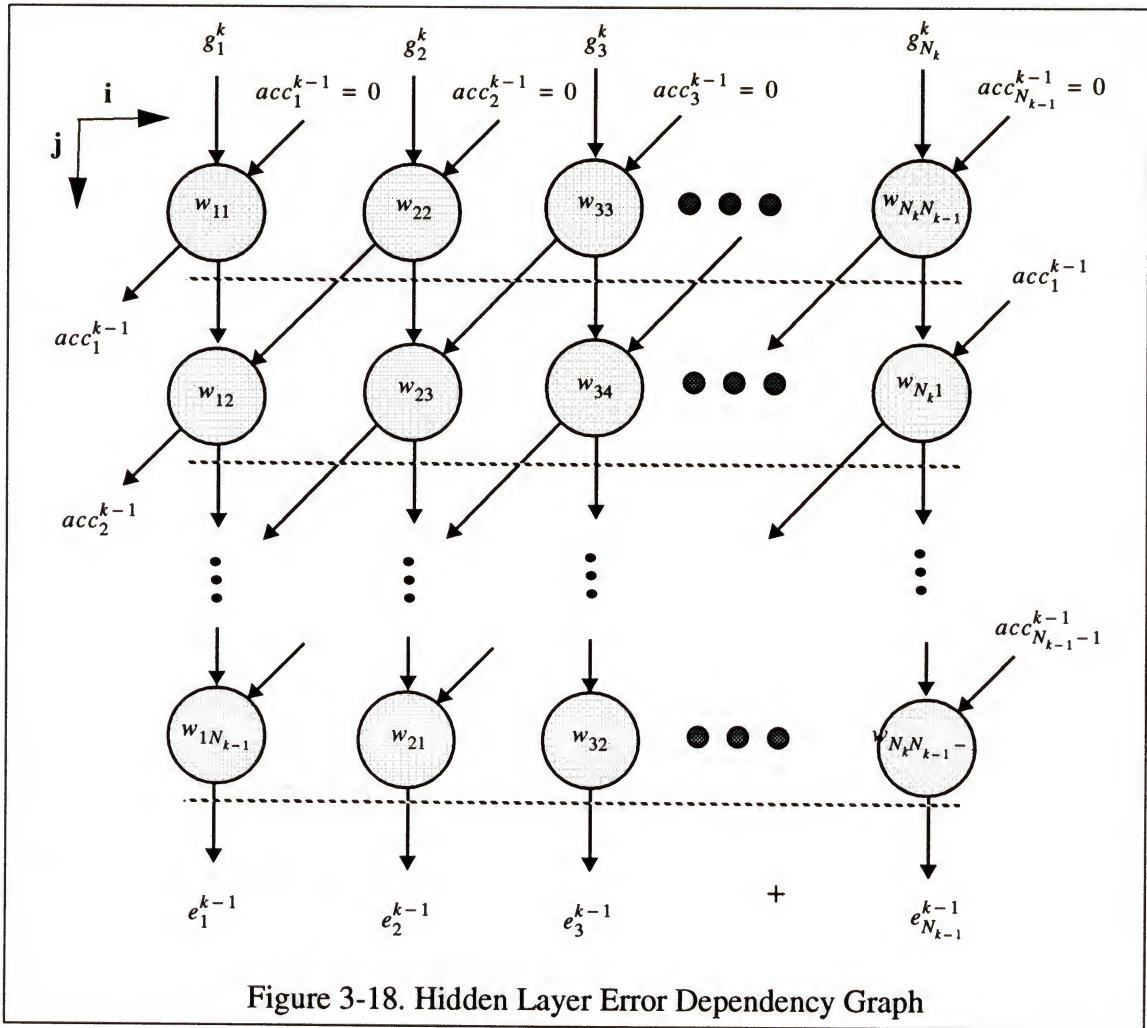
the hidden error was given as: $e_j^{k-1} = \sum_{i=1}^{N_k} w_{ij}^k \sigma'(u_i^k) e_i^k$ Because $\sigma'(u_i^k) e_i^k$ can be com-

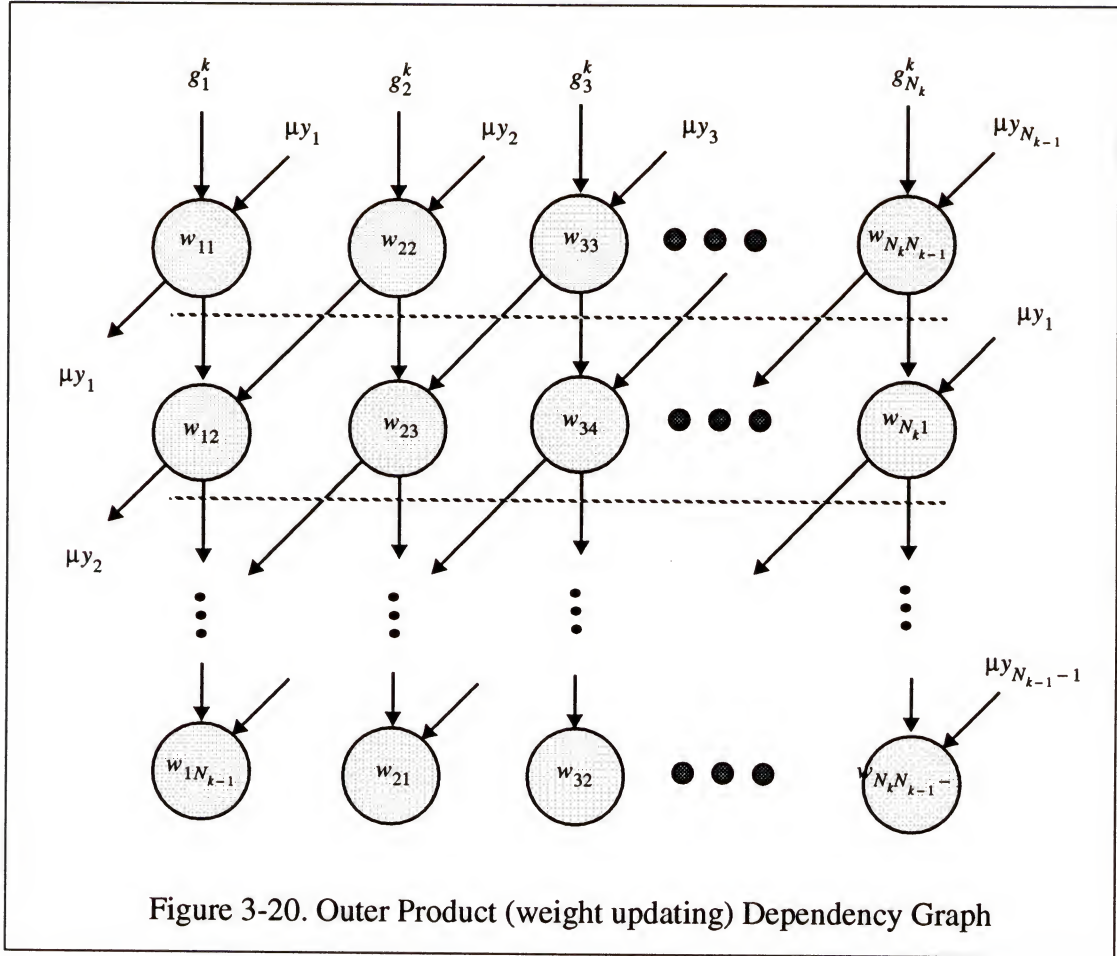
puted locally (all the necessary data for the calculation resides in the array unit) it is grouped as a variable, g_i . The hidden error will then be computed as an accumulation just as in the retrieval mode computation of the dot-product. This produces the new hidden layer

error vector equation, $acc_j^{k-1} = \sum_{i=1}^{N_k} w_{ij}^k g_i$. From this form, we can obtain the DG required

for vector matrix multiplication (hidden layer error calculation). See Figure 3-18. The outer error propagated back to the input side of the k PEs corresponds to the variable g_i . These inputs are shown as being passed from one node to another in the i direction from top to bottom. The i and j axis switch because the propagation is now from left to right in the network. The weight indexes do not change however, because they are referenced as before (only the propagation is changing direction. Inside the node, the g_i terms are multiplied with the weight and added to acc_j . Note: initially acc_j is set to zero. The scheduling planes (dotted lines) are the same as in retrieval case. The computation inside the DG node is shown in Figure 3-19. The accumulation must traverse along the diagonal because of the summation over i. Similarly, the g_i terms must be propagated vertically due to the weight ordering.

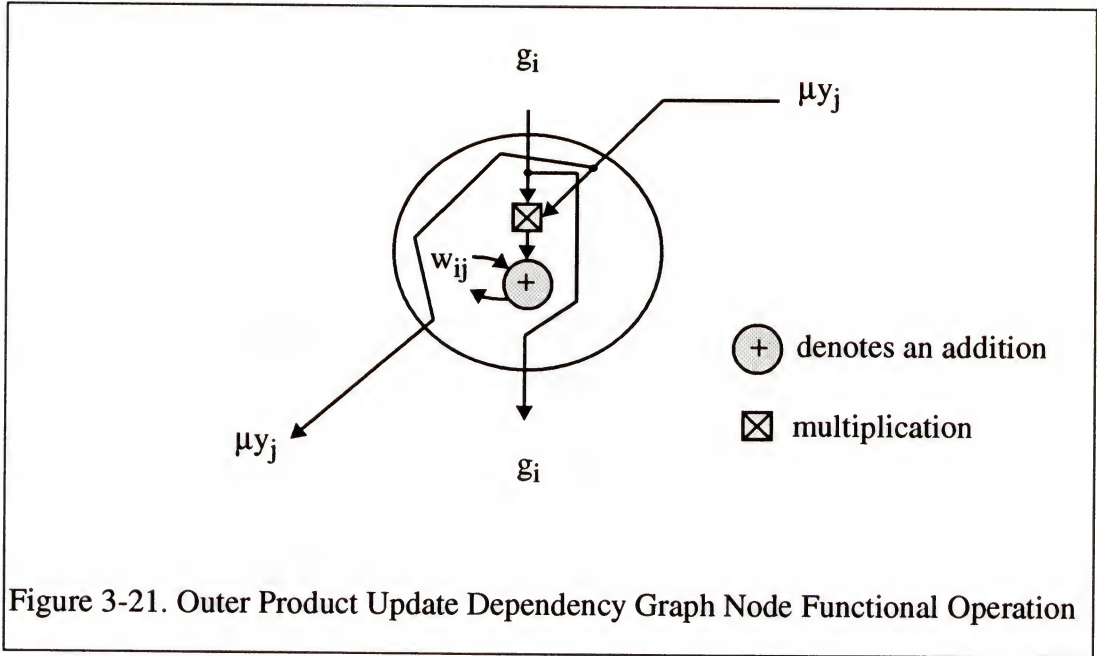
acc_j is denoted as acc_{ij} to show that after an i addition the new accumulation will correspond to $acc_{i+1,j}$ (the j index value does not change after an operation, it is constant throughout the DG). Once e_j^{k-1} has been computed, g_i^{k-1} can be computed locally which is equivalent to $\sigma'(u_i^{k-1}) e_i^{k-1}$ in equation (2-5), $\Delta w_{ij}^{k-1} = \mu \sigma'(u_i^{k-1}) e_i^{k-1} y_j^{k-2}$. The updating of the weights will now proceed.





The weights are updated in the following manner. $w_{new} = w_{old} + \Delta w = w_{old} + \mu y_j g_i$ The superscripts were purposely left out to generalize the weight updating equation (outer product updating). The DG and node function figures which correspond to this operation are shown in Figures 3-20 and 3-21. Again the positioning of the g_i terms and μy_j activations into the node is dependent on the ordering of the weights in Figure 3-20. It is also important to note that no accumulation is being passed, only the propagated errors and left most activation (with respect to the weights that are being updated) are transferred.

Returning to the example where there were 4 inputs, 12 hidden layer PEs, and 8 output PEs, the systolic array backpropagation will now be discussed. As before, the number of array units will be chosen to be four for the same reasons mentioned in retrieval.



Recalling that $g_i = \sigma'(u_i^k) e_i^k$, g_1 - g_8 are first computed in parallel among the 4 array units. Once the g_i error terms are computed, the hidden error computations may commence. This will appear very similar to retrieval processing on the ring array. See Figure 3-22. The 12 final accumulations, acc_{1-12}^1 , are equivalent to the hidden layer errors, e_{1-12}^1 . Unlike the retrieval ring array processing, the accumulations are passed around instead of remaining resident in the array unit. The accumulations are split into three groups of four values and are cycled clockwise in a similar manner as the inputs in retrieval. One cycle again consists of shifting an acc_j sum four times. acc_{1-4} are cycled once while the g_{1-4} are multiplied with the upper left column of weights under an array unit. This will compute half of the sum for a particular acc_j accumulation. The other half of the sum is computed when acc_{1-4} are again cycled and g_{5-8} are multiplied with the upper right column of weights under the corresponding array unit. This also repeats for acc_{5-8} ; where they use the middle two columns of weights. Finally, acc_{9-12} are cycled twice to yield all 12 hidden error values. A total of six cycles was therefore required to compute all the accumulations. This completes the vector

matrix multiplication portion of backpropagation and weight updating is next. The output layer weights will be computed first. Recalling that $w_{new}^2 = w_{old}^2 + \Delta w_{ij}^2 = w_{old}^2 + \mu y_j^1 g_i^2$, the g_i error terms will remain stationary in the array units. See Figure 3-23. The 3 groups of μy_j terms are cycled twice each to update the left and right columns of weights. Therefore, 6 total cycles are needed to update all the output weights. A cycle equates to shifting the data around the ring four times (complete transversal around the ring). Figure 3-24 illustrates the hidden layer weight updating which will require 3 total cycles.

3.7 Chapter Summary

While analog hardware based systems offer extremely fast neural computations rates, there are several disadvantages to using one of these systems as a general purpose research tool. The first is activation and weight resolution. Most systems offer only 5-8 bits of accuracy and this may affect (degrade) the learning mechanism or network one is testing. In other words, the system implementation may cause a failure rather than the algorithm or procedure being tested. An example is that a particular neural net classification may train well on a high precision digital machine and never converge on an analog system.

Other problems inherent in analog systems are their ability to expand and lack of computational flexibility. Because analog systems are usually built out of dedicated custom hardware, if the neural application doubles in size, the hardware must increase proportionately. In a digital system the problem can be re-mapped onto the existing hardware (at a cost of loss in computation speed). Analog systems are also designed to perform a specific set of functions. Preprocessing (e.g., low pass filtering, spectral or power feature/attribute extraction) therefore must be performed on a another machine. However with a digital system, both the preprocessing and neural net functions may be able to exist on a single system.

One last problem with analog systems is that they are usually very expensive (\$20K and up). Where they may be cost effective though is in mature ANN applications with trained weights. This would be a retrieval mode only system where a custom VLSI chip(s) would be designed to perform high speed classification for a unit designated for mass production. Large unit/chip quantities could therefore make the custom VLSI implementation

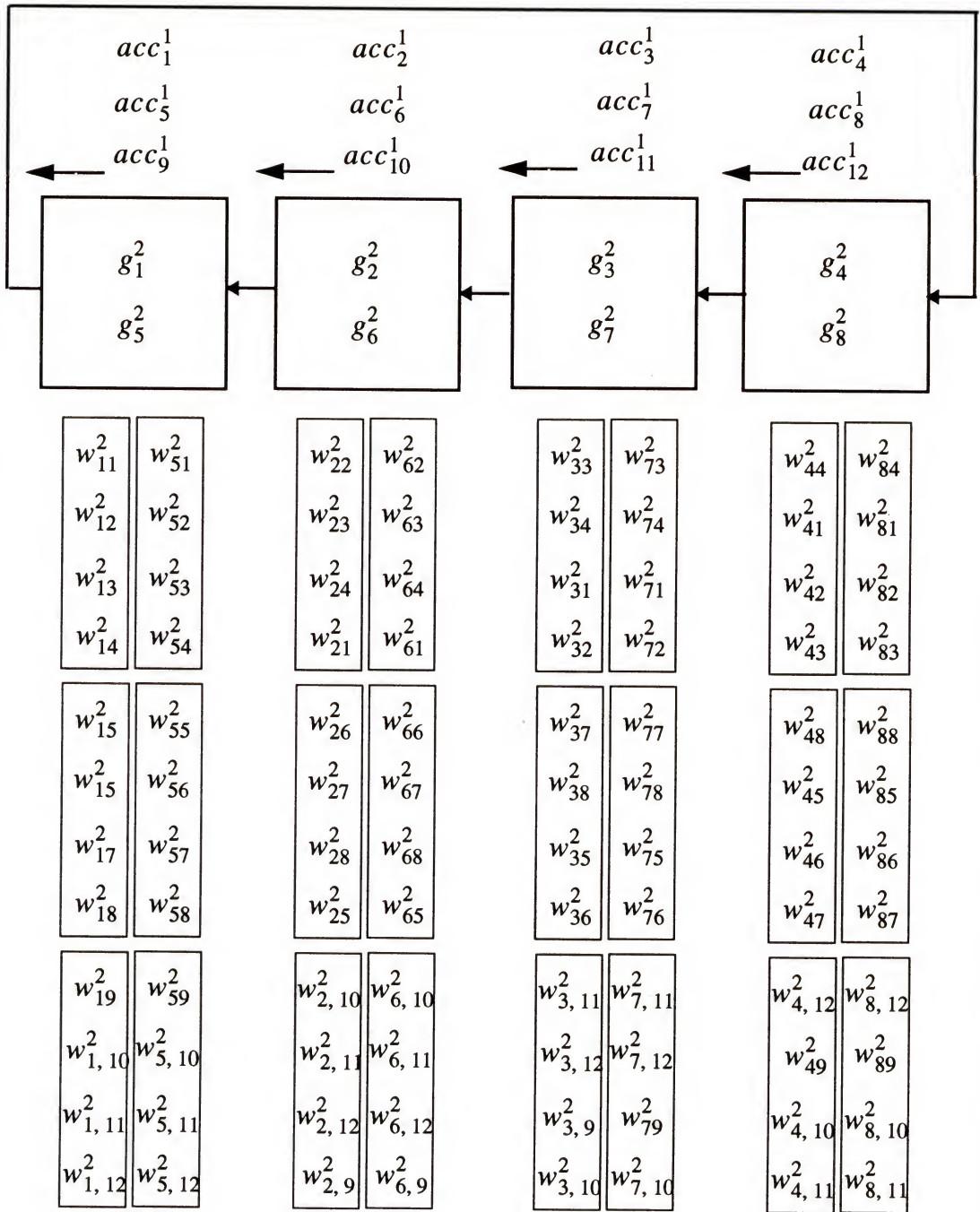


Figure 3-22. Hidden Error Computation (4-12-8 Neuron Network)

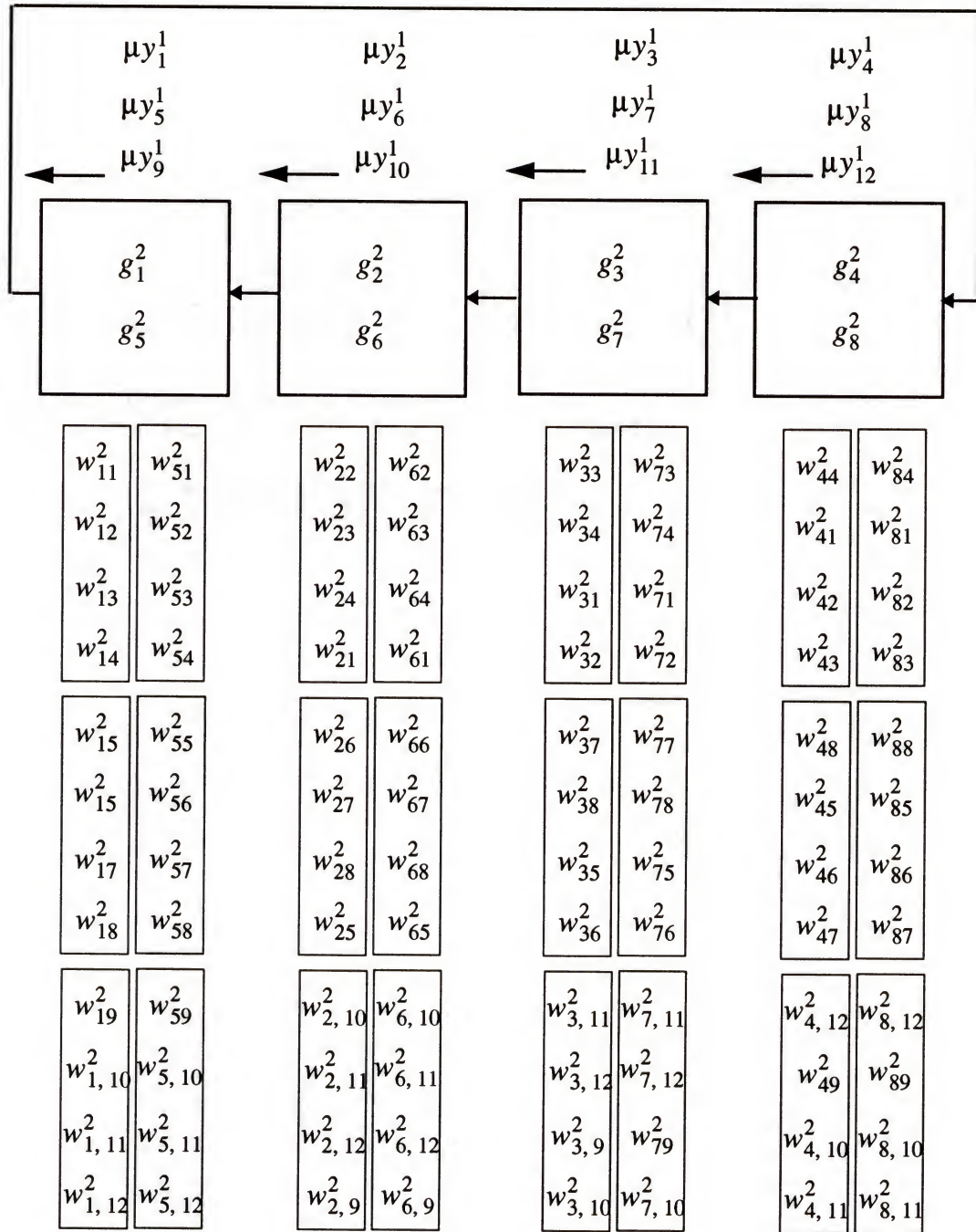
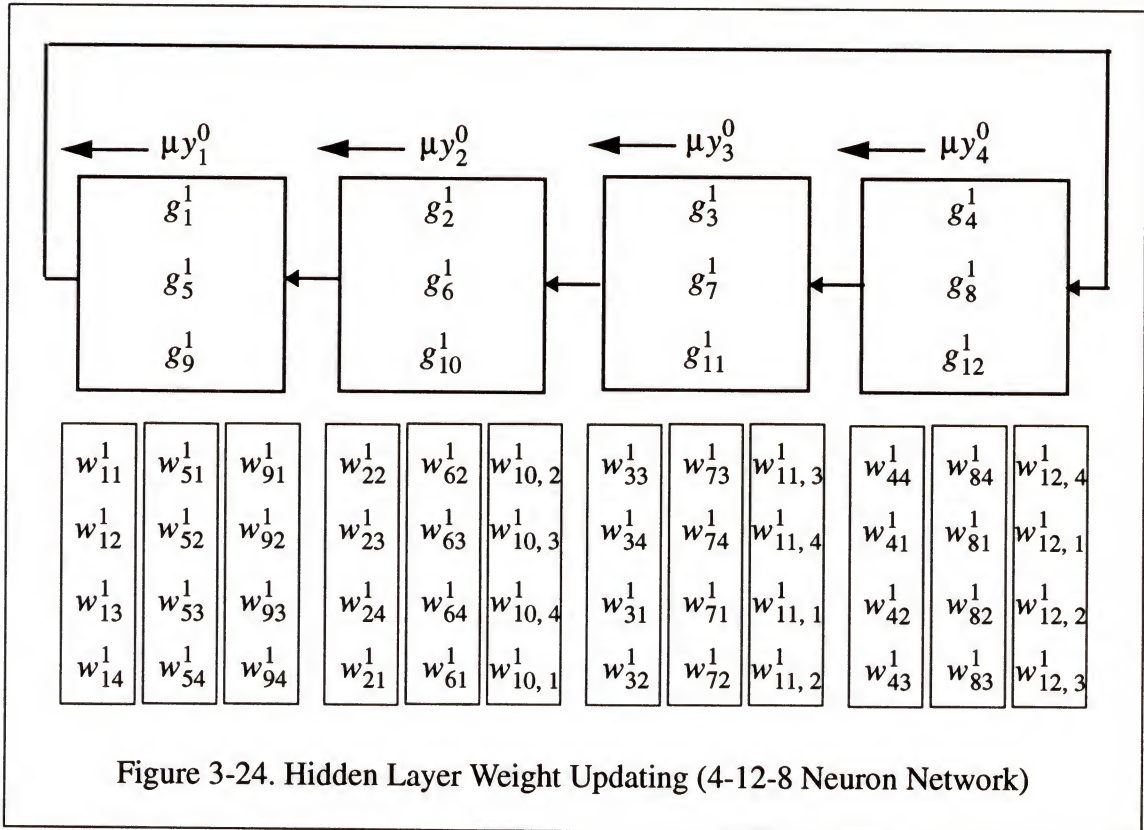


Figure 3-23. Output Layer Weight Updating (4-12-8 Neuron Network)



affordable. In summary, analog systems may offer excellent solutions for specific applications but are expensive to use for general purpose artificial neural net research.

Digital systems sacrifice speed for both scale-ability and computational flexibility. Different size problems can be easily re-mapped onto existing hardware. Coarse parallel systems (e.g., clusters of DSPs, ALUs, or work stations) contain a base processing unit which can perform many different types of operations (extensive instructions sets). Fine grain and array processor based systems trade flexibility for additional speed. These systems usually contain several hundred to many thousand custom processors. This greatly increases parallelism but decreases functionality. Custom processors/bus structures are not 'off the shelf' components, therefore another problem is that economic cost greatly increases when compared to the coarse system. However this may be acceptable for a large

university or government research lab which requires extensive speed-up and is not as concerned with economic cost.

Systolic arrays and Neuro Turbo were presented in great detail to illustrate two types of static network partitioning schemes. In both cases, inputs, weights, and desired signal elements were evenly distributed across the system processors. Neuro Turbo passes partial nets around its ring during retrieval while systolic arrays pass the previous layer activations around the array. In either case, *the global communication grows with respect to N (the number of PEs in a layer) while local processing grows with respect to N^2 .*

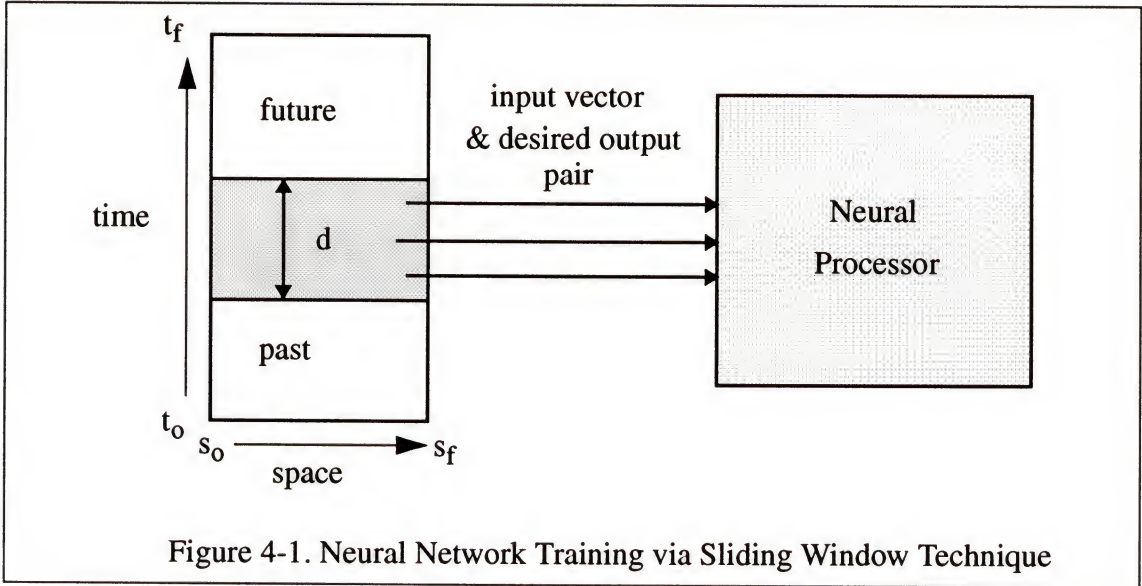
During backpropagation Neuro Turbo passes error terms through the ring while the systolic array sends error accumulations (e.g., sum of errors backed through the weights). As in retrieval, the global communication overhead for both systems grows with respect to N . This should then again be compared to the local computation (e.g., retrieval dot product, error backpropagation through wts., and weight updating) time which grows with respect to N^2 . In conclusion, both partitioning schemes offer excellent storage distribution and reduced global communication.

CHAPTER 4 THE GAMMA MODEL: A DYNAMIC NEURAL NETWORK

4.1 Introduction

Dynamic neural nets are a temporal extension of the static network discussed in Chapter 2. While a static neural network contains no feedback paths, a dynamic network is slightly more complex in that it contains some type of short term memory structure [35]. In the dynamic model, a memory structure of some type is concatenated on to the model in order to eliminate the problem of input data segmentation found in the static case. For example, suppose we are given a simple neural processor (i.e. multi-layer perceptron previously discussed) as shown in the right side of Figure 4-1. A window or input vector, shown on the left side of Figure 4-1, is presented to the net for training. Thus, the vertical axis corresponds to the temporal dimension and the horizontal axis is a spatial dimension which corresponds to some computation. The window width, d , equates to a group of discrete patterns at a particular instant in time and has an associated desired output which will be used to train the network. The spatial axis, for example, could be the spatially different electrode values of EEG waveform. The window, therefore, could be hundreds of these vectors over time which correspond to a set of brain wave dynamics one desires to classify. In this crude example, the network may be trained to identify spike detection in EEG signals using a set of known training patterns.

Continuing on with this example, the first problem that arises is what window width should be selected for the application. If the window width is made to encompass the entire temporal dimension t_0 - t_f , the number of weight vectors required by the net adequately learn the desired dynamics becomes prohibitively large. This in turn creates a lengthy learning

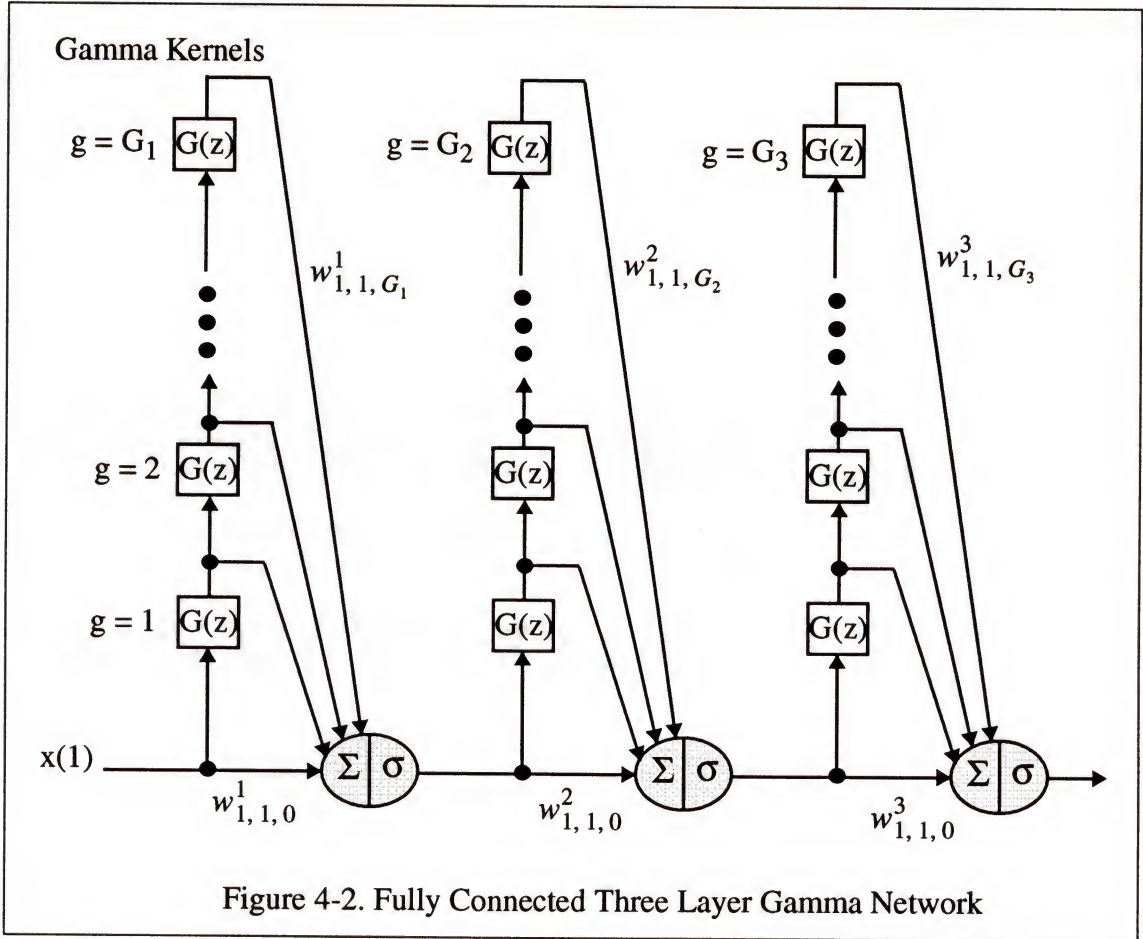


time; or more concisely stated, the neural net learning time scales worse than proportional with the dimension of the weight vectors [35]. On the other hand, if d is made too small, the net may not be able to learn the signal dynamics one desires. This information may be spread out over the entire temporal dimension. Thus the most common solution is to slide the window through time such that the network receives all training vectors but only d many at a time (training period). This method, commonly employed, has a second problem associated with it; this is the problem of trying to track non-stationary signals. A large d shifted through time will tend to average the time varying statistics of the signal and a small d will make the classification very sensitive to the particular signal segment (window) being utilized. In summary, the two main problems are setting the proper window width and how the windows are to be weighted in comparison to one another. The dynamic neural net addresses both of these problems in an adaptive manner. The input data adapts both the effective window width and strength in weighting of the training set.

A short term memory model known as the gamma model attempts to overcome these inherent static neural net windowing problems [35] [36]. The term ‘static’ thus refers to the notion that the activations are a function of the current window only. Dynamic suggests that the activations are functions of previous windows (training sets). Although there are other dynamic neural models which address these problems, the gamma network was specifically chosen for analysis and partitioning because of the level of expertise available in the CNE Lab regarding this model and also due to the overwhelming desire of lab personnel to speed-up their gamma related applications.

4.2 Topology and Retrieval

The gamma neural model is illustrated in Figure 4-2. This is an extension of the static model shown earlier in Figure 2-3. In Figure 4-2, the bottom layer is comprised of the fully connected two layer static network. The static network has been laid flat such that the inputs, PEs, and weight layers (which were parallel to the vertical axis in Figure 2-3) are now parallel to the axis which is perpendicular to the paper. The short term memory is implemented via the gamma kernels, $G(z)$, which are now in line with the vertical axis. The superscript again is used to denote the layer and a third dimension is added to the weight vector to identify that it is a member of a particular gamma kernel. Hence for every PE in the network, there is a corresponding group of gamma kernels. The number of these kernels can vary from layer but is constant among the PEs in a given k layer. For example in Figure 4-1, there are G_1 gamma kernels in the first layer, G_2 kernels in the second layer and so forth. The gamma kernels can be thought of as stacking several planes of static networks on top of the $g = 0$ level static network. A *focused gamma network* is a subset of the gamma model where only the input PE layer has gamma kernels. In other words, there are no gamma kernels stacked on the output layers or hidden layers. See Figure 4-4. This is a popularly used algorithm in the CNE Lab.



The function of an individual gamma kernel is shown in Figure 4-3. The input is brought in through the bottom and multiplied by the parameter γ and then is summed with the feedback term from the next state in a manner similar to a locally recursive IIR filter. In fact, if γ is made 1, the gamma kernels form a simple tapped delay line. In Figure 4-3, the input to the $g = 1$ kernel is either an input (if $k = 1$) or an activation from the previous layer ($k > 1$). The j index corresponds to the particular PE being addressed in the given k layer.

Given a G_k order gamma kernel structure, where $k = 1, 2, \dots$, the no. of layers in the

network, the memory depth D is given as: $D = \frac{G_k}{\gamma}$, γ is an adaptive parameter and is

$$G(z) = \frac{\gamma}{Z - (1 - \gamma)}$$

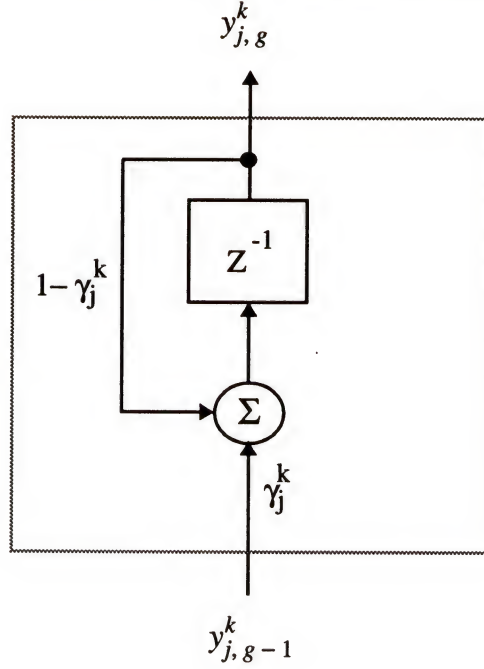


Figure 4-3. GammaKernel, $G(z)$, Definition

termed the memory resolution. The kernel order, G_n , is selected a priori by the user. The input/output relationship created by the discrete gamma kernel is best described with equation (4-1). In (4-1), the $t-1$ term refers to using inputs from the previous state to compute the present state output. Because of the recurrent nature of this equation, the previous input

$$y_{j,g}^k(t) = (1 - \gamma_j^k) y_{j,g}^k(t-1) + \gamma_j^k y_{j,g-1}^k(t-1) \quad , \text{ for } g = 1, \dots, G_k \quad (4-1)$$

value (or activation depending on k) and all gamma kernel values from the last state must be stored in memory. This is a major difference from the static neural model. In the static multilayer perceptron no information is saved or used from a previous state (other than the updated weights). In the focused gamma net, information from the past (or events occurring

through time) must be used to train the network. Thus the net is more suited for classifying time varying signals. For simplicity, the dependence on t will be dropped from the equations.

Once these kernel outputs are computed in (4-1), they are multiplied by their appropriate weight vectors, summed, and again sent through the squashing function. See the gamma retrieval equation (4-2). It is important to note that the y terms on the right side can be either activations from the previous layer or gamma kernel outputs (which is not termed an activation). Note: The (t) has been dropped from the activation shown in (4-1) for simplicity purposes. However it should be pointed out that all activations are functions time. Equation (4-3) illustrates the generation of the net required in (4-2).

$$\text{Gamma Retrieval} \quad y_{i,0}^k = \sigma \left(\sum_{j=1}^{N_{k-1}} \sum_{g=0}^{G_k} w_{ij,g}^k y_{j,g}^{k-1} \right) \quad (4-2)$$

$$\text{Net Generation} \quad u_{i,0}^k = \sum_{j=1}^{N_{k-1}} \sum_{g=0}^{G_k} w_{ij,g}^k y_{j,g}^{k-1} \quad (4-3)$$

In the focused gamma net there are kernels only at the input layer PEs. Hence (4-2) is valid for computing the input layer activations, but reverts back to retrieval eq. (2-1) for generation of $k > 1$ layer activations. The gamma model accepts memory kernels in any network layer. Here we will restrict the discussion to the most widely used topology, the focused gamma net, but the partitioning scheme applies to any implementation of the gamma model. For a more in-depth discussion of the gamma model and its derivation, consult B. DeVries Ph.D. dissertation and C. Lefebvre masters thesis [37] [38].

As described in the last chapter, the focused gamma net was partitioned across a cluster of NeXT work stations for neural net experimentation rather than the gamma topology shown in Fig. 4-2. This was chosen because of a deeper understanding of the focused

gamma net versus the gamma model. As mentioned earlier, in the focused gamma net we shift the input through the gamma kernel to extract time varying signal information. Depending on the memory depth, i.e. number of gamma kernels, the focused gamma net can be trained to store (or classify) a length of time varying signal.

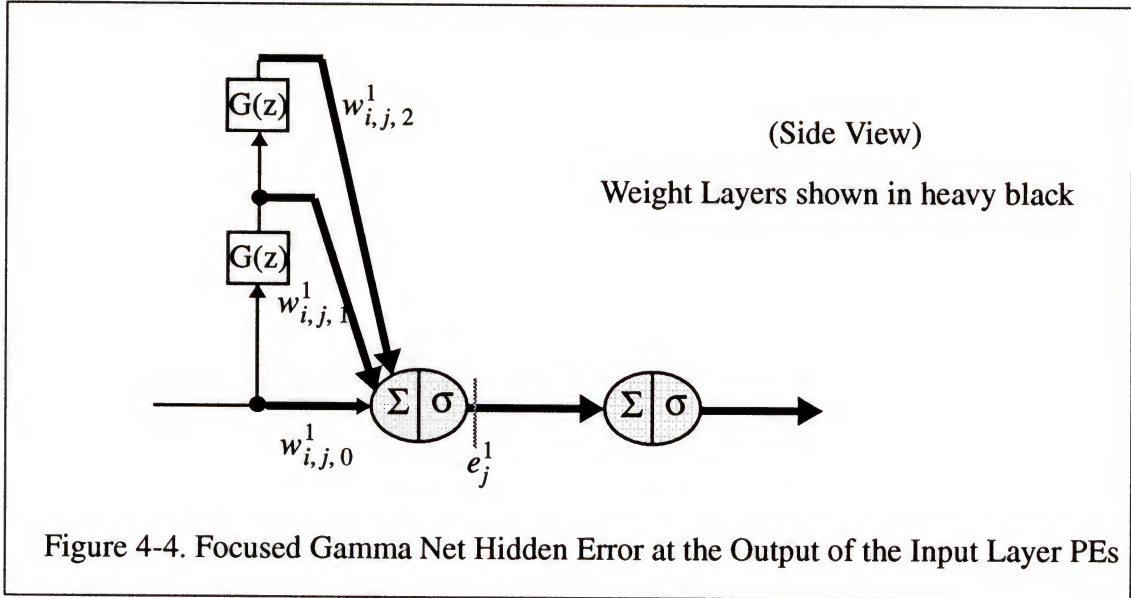
4.3 Backpropagation in Recurrent Networks

Backpropagation through the focused gamma network has computational similarities to that found in the static multi-layer perceptron. After completing a retrieval cycle, the output error is first determined as in equation (2-3). Similarly, the error is propagated left through the network, as in backpropagation for a static network, for all layers of PEs not containing gamma kernels. See Figure 4-4. Equation (2-4) once again applies to this operation. Once an error term has been obtained at the input layer PE (e_j^1 in Figure 4-4), it is again propagated through the PE by multiplying it by the derivative of the non-linear transform. This new error is then used to update all the input layer weights and gamma layer weights as shown in (4-4) and (4-5). If the sigmoidal non-linear transform is again used, $\sigma'(u)$ again can be replaced with $y(1-y)$ as before. Where y is the activation generated by the input layer PE. In equation (4-4) it is also important to note that y_{jg}^0 is either the input value x (when $g=0$) or a delayed input (when $g>0$). Equation (4-5) illustrates that the weights are updated as in the static case described by (2-7).

$$\begin{array}{ll} \text{Focused Gamma Net Delta Weights} & \\ \text{(Input Layer)} & \Delta w_{ijg}^1 = \mu \sigma'(u_i^1) e_i^1 y_{jg}^0 \end{array} \quad (4-4)$$

$$\begin{array}{ll} \text{Focused Gamma Net Weight Update} & w_{new}^k = w_{old}^k + \Delta w_{ijg}^k \end{array} \quad (4-5)$$

Hence very little has changed in the error backpropagation and weight update process for the weight layers of the focused gamma net which do not contain gamma kernels. In the gamma weight layers however, the gamma kernel output is multiplied by the error



(shifted through the PE) to obtain a delta gamma weight value. Thus the weights are being adapted in relation to an input vector which is shifted through the gamma kernel.

The gamma parameter, γ , for a given gamma cascade can either be preset initially or allowed to adapt. Presetting the value may be done to force a particular condition onto the network for experimentation purposes. Otherwise γ is adapted in the following manner. The error propagated through the input PE layer, $\sigma'(u_i^1) e_j^1$, is propagated and summed through the static and gamma weight layer as in (4-6). This is equivalent to backing the errors at the output of the input layer PEs through the corresponding input layer weights. The result is an error sum at each side of the gamma kernels or $G(z)$ taps. This error is then multiplied by the input and delayed input as in (4-7). The gradient is then multiplied by the learning rate to obtain the gamma delta shown in (4-8). Equation (4-7) therefore approximates how the error surface changes with respect to small changes in the gamma coefficient. The new gamma coefficient is obtained by adding the computed delta to the old coefficient (the same process as updating the weights).

$$\text{Error at the Gamma Kernels} \quad e_{jg}^0 = \sum_{i=1}^{N_k} w_{ijg}^1 \sigma'(u_i^1) e_i^1 \quad (4-6)$$

$$\text{Gamma Error Gradient} \quad \frac{\partial E}{\partial \gamma} = \sum_{g=1}^{G_k} \varepsilon_{i,g}(t) [y_{i,g-1}(t-1) - y_{i,g}(t-1)] \quad (4-7)$$

$$\text{Delta Gamma} \quad \Delta \gamma = -\mu \frac{\partial E}{\partial \gamma} \quad (4-8)$$

4.4 Parallelism in the Focused Gamma Network

The parallelism found in the focused gamma net is nearly identical to that described in Chapter 2 in sections 2.3.1, 2.3.2, and 2.5 for the multi-layer perceptron. The only difference is that adding additional weight layers (gamma weights) increases the amount of possible parallel computation in the first layer of PEs during retrieval. In the case where a single weight is assigned to a single physical processing element, it is possible for all input/weight multiplications to be performed in parallel. However this again requires that processors be able to concurrently read the input and delayed input values. This also greatly increases the overall processor count.

In the alternate case where a single physical processor is mapped to a single PE, the gamma model will appear to have increased efficiency as compared to the static network. This is due to the comparison between local computation of a processor versus its global communication. For example in this one to one PE processor mapping all processors would receive inputs and begin computing. In the gamma case all processors first must retrieve previous time inputs (unit delayed inputs) to generate the gamma kernel inputs. This is a local operation. Upon generating inputs, all processors can then begin retrieval and generate the partial sums (nets) which correspond to all the gamma weight layers. This is again a local operation which is G (number of gamma kernels) times the static retrieval operation.

Once the partial sums are completed they can be locally transformed to yield the activation. These values will then have to be globally communicated to the processors assigned to computing the activations in the hidden layer. Hence the local processing has increased G fold in the gamma case while communication stays the same as compared to the static network. The efficiency in parallelism therefore looks better due to the increase local computation time when compared against communicating values globally. A similar argument can be presented for backpropagation of the errors and weight updating; the local processing has increased but the global communication (error terms) will remain the same when compared to the static network. In conclusion, the processing between layers (retrieval dot product, backpropagation of error) increases by G per layer containing gamma kernels while the communication of inputs and activations (required by the next layer of PEs) stays the same as in the static net case.

4.5 Partitioning Locally Recurrent Neural Networks

Partitioning a locally recurrent neural network is very similar to the process described in section 3.5.1 for partitioning a multi-layer perceptron. However before this is presented, a brief discussion of what constitutes a locally recurrent network is given.

The gamma model and its subset the focused gamma net are examples of neural nets with local recurrence. The feedback terms found in the kernels are local to gamma kernel structure and thus are local to a single input element or activation (depending on the layer in the gamma net). See Figures 4-2 and 4-3. Thus the recurrence is local to an input and/or processing element (PE) activation depending on which layer you are addressing in the Focused gamma net or gamma model. However if the feedback shown in Figure 4-3 originated in another kernel structure (residing on top of another input or PE), it would no longer be locally recurrent to a single input or PE. This is termed locally recurrent to a given layer of inputs or PEs. Expanding this further, we arrive at the general case where feedback can be

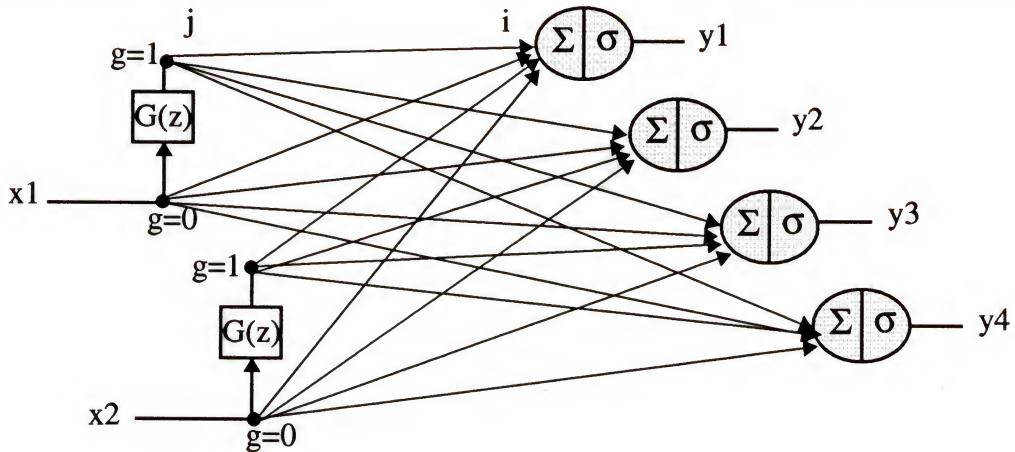


Figure 4-5. Partitioning the Focused Gamma Net, Example Network

globally recurrent. In this type of system, feedback can originate and terminate in different inputs/PEs throughout the network. In this type of topology the concept of layers vanishes. As a simple example, activations in layer $k=2$ and those from layer $k=3$ could be used as feedback to a PE in the original $k=2$ layer. There are no longer any constraints on where the feedback originates and terminates.

For simplicity purposes, partitioning the focused gamma net will be discussed first and then developed further to include the gamma model. Upon completing this, suggestions will be made regarding the partitioning a globally recurrent system.

Because the feedback is local to the kernel structure in the focused gamma net, the gamma weights and kernel computations will be partitioned in a similar manner as the MLP. The weights will again be partitioned according to backpropagation alignment and is best described with the following example. See Figure 4-5. In this figure there are two inputs, one kernel per input and four output PEs. Assuming that this will be partitioned on a two processor system ($P=2$), the wts. are divided into the local memory of the processors

as follows. Beginning with the $g=0$ weight layer, processor A will contain weights $j=1, i=1-4$ while processor B contains weights $j=2, i=1-4$. This is similar to the partitioning shown in Figure 3-4. The $g=1$ layer weights will also be partitioned as the those in the $g=0$ layer. Processor A will contain weights corresponding to $j=1, i=1-4$ and B will locally maintain weights $j=2, i=1-4$. Thus with this weight partitioning scheme, Processor A will receive input x_1 and B will get x_2 . The kernels associated with x_1 and x_2 will also be partitioned into processors A and B, respectively. Processor A will then compute activation y_1 and y_2 while B will compute y_3 and y_4 . For this, partial nets will need to be passed between processors as described in section 3.5.1 (static case).

Summarizing the partitioning, all layers of gamma weights partition just as the $g=0$ layer. The $g=0$ weight layer are the weights common to the static network and thus are partitioned according to Figure 3-4. Inputs and layers of PEs divided evenly among the system processors as in the static neural network partitioning. The important point is that the gamma structures ($G(z)$ blocks) follow the partitioned inputs. A gamma structure associated with a single input is never divided (kernels split up) amongst the system processors.

When compared to the MLP, the partitioned gamma kernel structures and gamma weights will require additional local memory for a system processor. For storage of the kernel outputs, each processor will require GN/P additional storage locations. Where G is the number of gamma kernels in a single structure and N the number of inputs. For the gamma weights, local storage will have to increase by GN_1N_2/P locations. Where N_1 is the number of inputs to the net and N_2 is number of PEs in the layer where the gamma weights terminate.

In addition to requiring more storage space, additional computation time will also be required for computing gamma kernel outputs, propagating the kernel output through the gamma weights, backing the error through the gamma weights and updating the gamma

weights. This is specifically quantified in the next chapter. Instead it should be pointed out that all the previous mentioned gamma related operations are performed locally by a system processor. All the necessary information (previous kernel values and required gamma weights) resides local to a processor and thus no additional global communication is required. *In fact, the global communication for the focused gamma net is exactly the same as that in the partitioned multi-layer perceptron case.* As in the MLP, only partial sums during retrieval and errors during backpropagation are passed between processors. The number of partial sums does not change for the focused gamma net. The processor simply adds in the new gamma dot product values to the partial net that was computed for the weight layer associated with an external input. (This is the $g=0$ layer in Figures 4-2, 4-4 and 4-5.) In summary, because the local processing increases while the global communication remains constant, implementing the focused gamma net on a parallel system will have a higher degree of parallelism as compared to implementing a multi-layer perceptron.

In the gamma model, where kernel structures exist in the hidden and output layers, the backpropagation alignment partitioning scheme can also be employed. The $k>1$ gamma weight layers and gamma kernel structures are partitioned as the $k=1$ focused gamma net gamma weights and kernels. See the k layer superscripts in Figures 4-2 and 4-4. Again this type of partitioning will contain the same amount of global communication when compared to the partitioned MLP consisting of a similar topology (minus the extra layers of gamma weights and kernels). However local processing will greatly increase (G times per k layer) and thus the efficiency or level of parallelism is expected to increase even further over the focused gamma net implementation.

In the case where global recurrence exists, the partitioning becomes far more complex. The backpropagation alignment scheme may or may not work depending on the number of recurrent connections and where they occur. For example, if the connections are

between activations which reside in a processor's local memory space, the previous state values don't have to be passed globally and parallel efficiency will be high. However if feedback connections are made between PEs which are assigned to different system processors, the previous state feedback terms will have to be transmitted as global data. This reduces the level of parallelism in the system. Further discussion of partitioning a globally recurrent network will be addressed in the end of this dissertation in the future work section.

CHAPTER 5

THE APPLICATION OF ATPD TO A GIVEN ALGORITHM AND ARCHITECTURE

5.1 Introduction

This chapter illustrates the method for applying ATPD to an algorithm designated to operate on a well defined system. In order to partition a particular algorithm onto a given architecture, as exemplified in Chapter 4, the algorithm should first be written in a form which highlights the basic operations comprising the process. This is performed first for the multi-layer perceptron operating in retrieval mode found in Chapter 2. See equations (2-1) and (2-2). These equations were also analyzed for pipeline-ability and parallelism back in sections 2.3.1 and 2.3.2. Once an in-depth understanding of the algorithm has been achieved, a tool referred to as algorithmic timing parameter decomposition (ATPD) can be applied. This is performed first using a *basic serial computer model* as the initial architectural constraint (benchmark). The architecture is then expanded to include pipelining and eventually parallel processing. Both the algorithm and the system will impose constraints on the ATPD analysis and results. Upon completion of the ATPD analysis of retrieval, the learning process (backpropagation) is decomposed.

This chapter also presents three ways of measuring the performance of a particular system. The three measures are the system's *computational bandwidth*, *cost* and the *order of growth in the computation time* as the network increases in size. The first number yields a raw speed estimation. The cost then takes into account both the computation time and the number of processors required. This gives a rating of the overall efficiency of the system. The final measure of merit tells a user how the cost and computation time grow with respect

to the network (problem) size. This measurement is performed by applying the Upper Boundary Theorem to the cost and computation time functions. These measurement techniques and algorithmic timing parameter decomposition (ATPD) are adaptations of work performed by S.G. Akl and S.Y. Su [16] [39].

One benefit which arises from applying ATPD to an algorithm using various different architectural constraints is that many different types of systems (analog or digital) can be compared to one another in terms of speed of processing. Speed ratings can then be computed for the various technologies discussed in Chapter 3. Because the analysis is directed at an atomic level, the specifics governing an improvement or deterioration in speed from one system to another can also be observed. Thus various different architectures can be rated and compared. In addition, improvements can be postulated for the selected architecture and/or partitioning scheme.

5.2 General Purpose Serial Computer

We begin the analysis with the most elementary processor model which has no parallelism or pipelining. This is referred to as a serial computer. Using this model as the architectural constraint, eliminates the idea of partitioning. There is only one processor and it can perform only one elemental operation in a discrete fixed amount of time.

The processor model is also expected to adhere to the following assumptions. The processor is assumed to fetch its instructions locally from memory, the instruction and data flow paths are separate and the processor contains some type of instruction cache mechanism. Under these assumptions, the instruction fetch time can overlap the data transfer cycle. This is due to the fact that because the computing tasks require few instructions (a multiply-accumulate operation), no new instructions need to be fetched into the instruction cache. Hence because the instruction execution is operating in parallel with the data transfer unit, the instruction fetch time can be ignored. The launch time for the initial cache loading

is also assumed to be negligible when compared to the data fetch time and multiply-accumulate time for a net computation loop.

5.2.1 Speed

The amount of time required to produce one output vector in retrieval is shown in (5-1). In equation (5-1), the terms in the summation represent the amount of time it takes to fetch, compute and store all the activations in a given layer. The sum is then taken over $k=1$ to K_L layers. Where K_L is the total number of layers in the network. See Figure 2-3 and equation (2-1). The summation in (2-1) has been converted into elementary read, write, multiply and accumulate times. These parameters are then grouped to yield (5-1).

$$\text{Serial Computer Computation Time} = \sum_{k=1}^{K_L} N_k N_{k-1} (t_{\text{fetch}}) + N_k N_{k-1} (t_{\text{fetch}} + t_{\text{ma}}) + N_k (t_{\text{nlt}}) + N_k (t_{\text{storage}}) \quad (5-1)$$

where;

- K_L = total number of layers in the network
- N_k = total number of inputs or PEs in a given k layer
- t_{fetch} = time required to fetch one weight, input or activation from memory
- t_{ma} = time to perform a multiply and accumulate
= $t_{\text{multiply}} + t_{\text{accumulate}}$
- t_{nlt} = nonlinear transformation time
- t_{storage} = time to store one PE output

The first term in the summation corresponds to obtaining the previous layer's activation or the input vector if it is the $k=1$ layer of processors in the network. Recalling in Figure 2-3, it should be apparent that N_{k-1} activations or inputs must be fetched from memory for each PE activation computed. Therefore in order to compute N_k activations, $N_k N_{k-1}$ fetches must be performed. If we assume that the processor only has to read the N_{k-1} inputs once, the first term will simplify to $N_{k-1} t_{\text{fetch}}$. Under this simplification it is assumed

that once the inputs reside locally, fetching them later for a multiply-accumulate operation is either negligible or lumped into the t_{ma} time. An example is a digital signal processor (DSP) chip which reads an external array into its internal (local) RAM. Once the internal buffer is loaded, the fetch and execution portions of the multiply-accumulate operation are pipelined and lumped as a single discrete timing primitive (i.e. t_{ma}).

The second term in the summation corresponds to fetching the weights in the current layer and then performing the necessary multiply and accumulate calculations. Because unique weights are required for each activation computation in a layer, this term cannot be simplified. Although the input vector is used repeatedly, the weights needed for each PE activation computation are uniquely distinct.

The third term in (5-1) is the nonlinear transform which must be computed for every processing element in the current layer under computation. t_{nlt} can be a computation time for a routine to compute the transform using a power series formula or t_{nlt} can be the average amount of time using a table look-up method [6].

The last term in equation (5-1) is the time required to store the activations externally. The computational bandwidth is defined as the inverse of this computation time. Thus if this time is estimated to be 10^{-3} seconds, the computational bandwidth for retrieval mode is 1000 output vectors per second.

To determine the order of equation (5-1), the following Upper Boundary Theorem is required [16]. The function $g(n)$ is said to be of order at most $f(n)$, denoted as $O(f(n))$, if there are positive constants c and n_0 such that $g(n) \leq cf(n)$ for all $n \geq n_0$. Equation (5-1) is now expanded and rewritten in a series form as shown by (5-2).

Applying the Upper Boundary theorem to the simplified series representation shown in (5-2), the order of serial computer computation for a neural net implementation in the retrieval mode is given as $O(\max(N_k N_{k-1}))$. Thus, the computation grows with

respect to the maximum value of the product of two consecutive layers of PEs or inputs. This is also equivalent to the maximum weight layer in the net which can be denoted as $O(\max(W_k))$.

(5-2)

$$\begin{aligned} \text{Serial Computer} \\ \text{Computation Time} &= N_1 N_0 C_a + N_1 N_0 C_b + N_1 C_c + N_0 C_a + N_2 N_1 C_b + N_2 C_c + \dots \\ &\quad + N_{K_L} N_{K_L-1} C_a + N_{K_L} N_{K_L-1} C_b + N_{K_L} C_c \end{aligned}$$

where; C_a is a constant = t_{fetch}
 C_b is a constant = $t_{\text{fetch}} + t_{\text{ma}}$
 C_c is a constant = $t_{\text{nlt}} + t_{\text{storage}}$
 N_k is the # of PEs in a given k layer

5.2.2 Cost

The cost of a parallel algorithm is defined as the product of the computation time multiplied by the number of system processors [16]. See equation (5-3). Normally, cost is developed in terms of a generalized processor - time unit; thus no specific processor- time is given. For example, an algorithm is analyzed and found to have a cost function which grows with respect to $\log(\text{no. of inputs/no. of processors})$, the cost unit depends on the system implemented. i.e. a system based on N Motorola 68000s which can perform the parallel run time in t msec would show a cost of order $t \log(n/N)$ Motorola 68000 - msec while implementing the proposed algorithm. Therefore in our serial computer case, where the number of processors is one, the cost grows as the running time, $O(\max(N_k N_{k-1}))$.

$$\text{cost} = (\text{parallel running time}) (\text{number of processors}) \quad (5-3)$$

5.2.3 Reducing the Serial Model's Computation Time

By applying ATPD to retrieval and using the serial computer as the system constraint, it is possible to identify the basic time components associated with performing the

computation. Once these components are identified, one can begin formulating ideas to reduce or eliminate them. The improvements to be had are therefore dependent both on device technology advances and architectural complexity. Examples of technological advances are faster memory chips and processors. These improvements shrink the system constants t_{fetch} , t_{ma} , t_{nlt} , and t_{storage} , thus widening the computational bandwidth. Architectural complexity can similarly increase the computational bandwidth but will do so without altering the system constants. This research is concerned with both how a technological improvement can be quantified and how architectural changes and improvements will affect the previously discussed figures of merits. The affects of adding complexity are now described.

Because of the elementary nature of equation (5-1), it is possible to formulate numerous schemes for reducing the overall computation time. One such scheme is to improve the function of the basic processor. An example is to create a unit which can read both the inputs (or previous layer activations) and weight values concurrently. The first two terms in (5-1) can now be simplified to one term. Kung's systolic array unit has this function and the alterations to the serial computation time are presented in section 5.6.

Another scheme for reducing the overall computation time is to create a system comprised of several processors (parallel processing). These processors can then be applied in one of three ways for multi-layer perceptron retrieval. The first construct being that all PE activations from a given k layer are computed by one processor. Under this partitioning, a layer of PE activations is assigned to a single processor and thus there are K_L processors for a K_L layer network. The second method is to force all the processors in the system to compute in parallel the activations from a particular k layer. Given this partitioning scheme, the processors divide up the N_k activations for parallel computation. Then the next layer's activations are partitioned amongst the system's processors for processing and so forth.

Finally, the last partitioning method consists of a mixture of the first two. Several processors are assigned to compute in parallel the activation for a given layer and the results are passed to another group of processors assigned to the next layer. Thus there is parallelism in a layer and pipelining between the groups assigned to a particular layer.

Only the case where retrieval is pipelined through a single array of processors will be discussed in the following sections. This is described mainly to illustrate and familiarize the reader with ATPD. It is important to see how changing the architecture alters the process flow (scheduling of tasks) and generates changes in the original serial retrieval computation times. The other types of parallelism are not analyzed due to the concentration of this work towards learning. Retrieval is not nearly as computationally intensive an operation as learning and so the focus of this work is dedicated to speeding up backpropagation. Retrieval is however a much more elementary algorithm to illustrate the application of ATPD.

5.3 Expanding the Serial Computer (Retrieval Mode Only) with Pipelining

The basic model presented in section 5.1 is now expanded to include pipelining between the layers of the network. K_L processors are connected such that each processor computes all the activations of a given assigned layer. As mentioned in section 5.2, if we have K_L layers we need K_L processors to form the pipeline. Equation (5-1) is adjusted to conform to this new architecture and scheduling as shown by (5-4). Equation (5-4) is no longer a summation but is theoretically assumed to be the maximum k layer computation time from equation (5-1). As stated in the end of section 5.2.1, the maximum consecutive layer product, $N_k N_{k-1}$, directly equates to the maximum weight layer in the network, $\max(W_k)$. Hence the serial processor assigned to the largest layer of weights will have the longest computation time. The *speed-up factor* S_f is equivalent to the Serial Computer Computation time divided by K_L Processors pipeline throughput period. The computational bandwidth will also improve S_f times. This factor is proportional to number of layers in a net and the

(5-4)

$$\begin{array}{l} K_L \text{ Processors} \\ \text{Pipeline} \\ \text{Throughput} \end{array} = \max \left[N_k N_{k-1} (t_{\text{fetch}}) + N_k N_{k-1} (t_{\text{fetch}} + t_{\text{ma}}) + N_k (t_{\text{nlt}}) + N_k (t_{\text{storage}}) \right]$$

where;

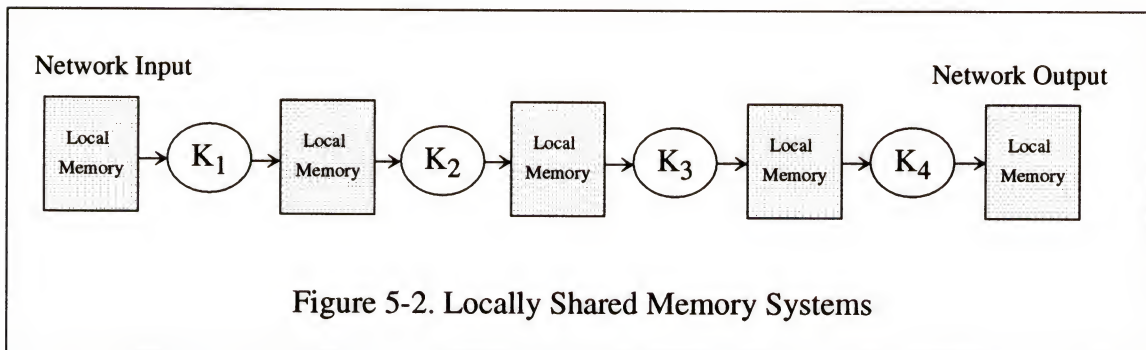
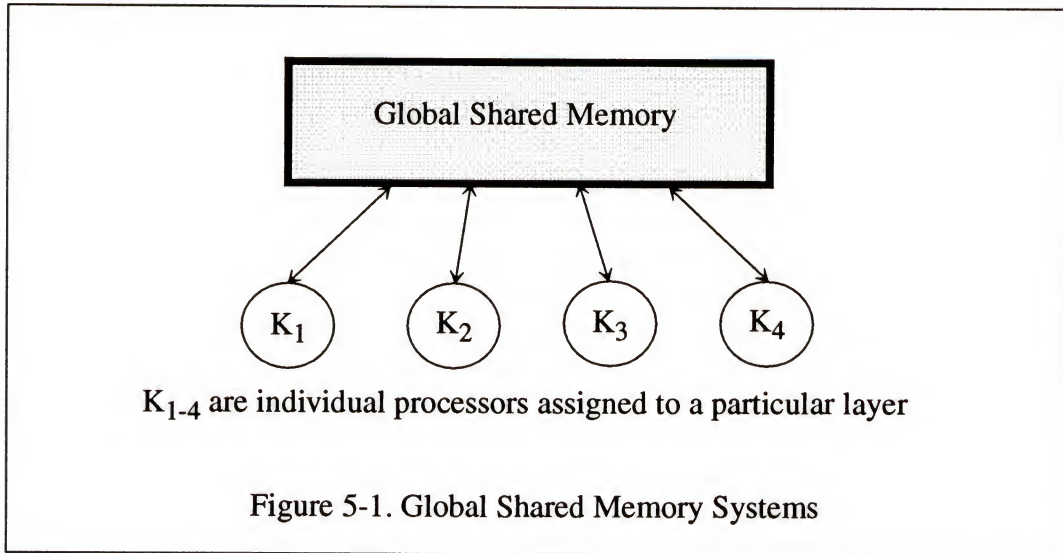
- K_L = total number of layers in the network
- N_k = total number of inputs or PEs in a given k layer
- t_{fetch} = time required to fetch one weight from memory
- t_{ma} = time to perform a multiply and accumulate
= $t_{\text{multiply}} + t_{\text{accumulate}}$
- t_{nlt} = nonlinear transformation time
- t_{storage} = time to store one PE output

relative sizes of each layer to one another. S_f will significantly increase if there are several layers of weights where all the layers of weights are approximately the same size. However, as the net weights in a layer approach infinity, the K_L processor pipeline throughput time grows with order $O(\max(N_k N_{k-1}))$ as in the serial case. The cost, however, grows as $O(K_L [\max(N_k N_{k-1})])$ which is significantly worse than the serial computer case. The computational bandwidth is widening at the expense of the cost function.

Also, in order for both the theoretical throughput period and the speed-up factor to remain valid, certain constraints must be met. These are discussed in the following sections and will turn out to be architecturally dependent.

5.3.1 Pipelining and Global Memory

K_L processors operating in a pipeline mode can communicate with each other in one of two ways, either by shared variable or message passing [40]. Equation (5-4) assumes a shared variable system with t_{storage} and t_{fetch} being the time to store and fetch data from a *globally shared memory* block. This architecture is presented in Figure 5-1. Sections of memory have been agreed upon from the start to hold a previous layer's outputs, weight



values, and the newly computed results. Memory can also be shared locally and is illustrated in Figure 5-2.

In the globally shared memory system, the layer processors must time share the global memory for both the fetch and store cycle if concurrent reading or writing is not permitted. Because of this non-concurrent read/write constraint, two other important factors must be adhered to if no additional time is to be added to equation (5-4). The first being a *processing vs. communication constraint* and the second a *synchronization issue*.

In a global shared memory pipeline system using K_L processors, the amount of time required for a particular k layer serial processor to fetch, compute and store its results is obtained directly from (5-4) and is shown as (5-5). This then can be broken down into (5-6) and (5-7). Thus in order for K_L processors to access mutually exclusive global memory and not add any extra time to the cycle, it can be shown that the total computation time for the slowest processor assigned to a layer (5-6) must be greater than all the other processors' fetch/storage times (5-7) summed together. The slowest processor will be the one assigned to the largest layer of weights. The other processors, having smaller assigned weight layers, should have shorter net computation times. This is best shown pictorially by Figure 5-3. The execution time found in Figure 5-3 corresponds to the time found in (5-6). Similarly, storage and fetch times in Figure 5-3 can be obtained from (5-7).

(5-5)

$$\begin{aligned} \text{Single Layer} \\ \text{Execution Cycle Time} \end{aligned} = N_{k-1}t_{\text{fetch}} + N_k N_{k-1}t_{\text{fetch}} + N_k N_{k-1}t_{\text{ma}} + N_k(t_{\text{nlt}} + t_{\text{storage}})$$

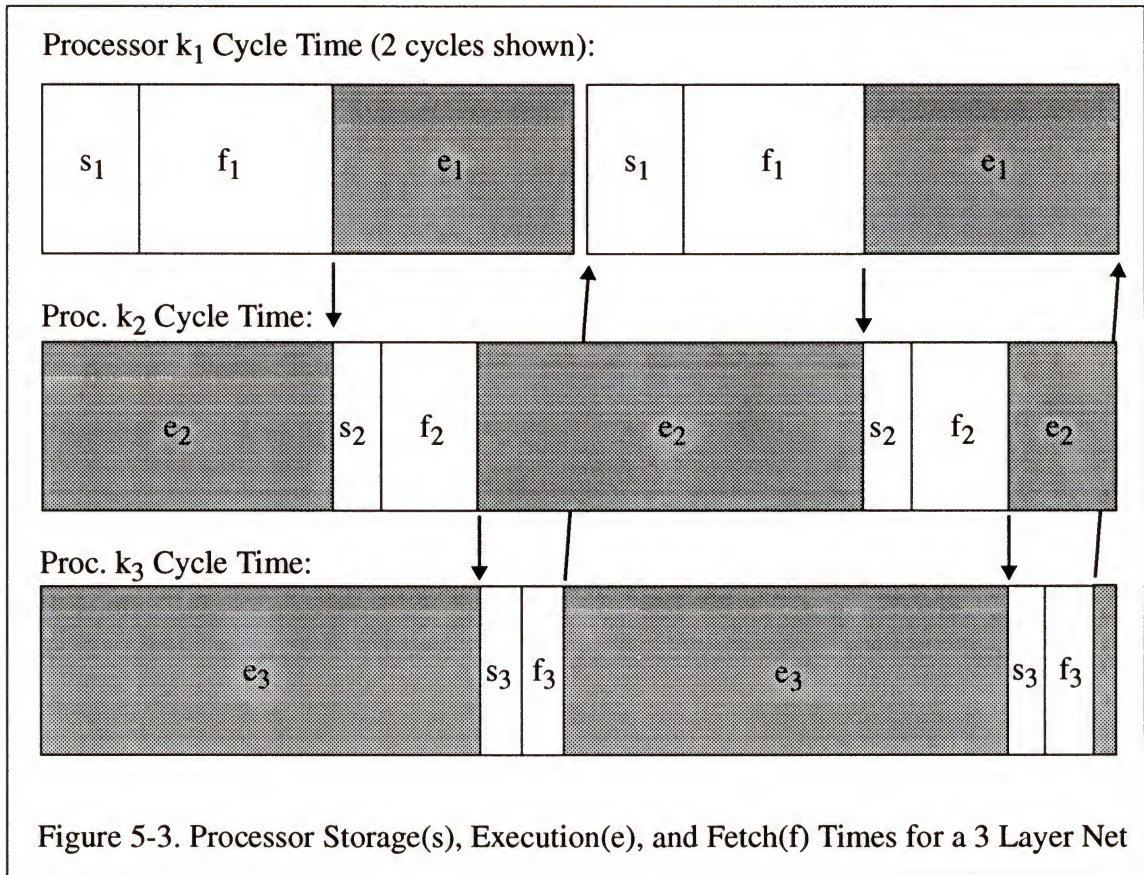
where;

$$\text{Total Processor Execution Time} = N_k N_{k-1}t_{\text{ma}} + N_k t_{\text{nlt}} \quad (5-6)$$

$$\text{Total Storage/Fetch Time} = N_{k-1}t_{\text{fetch}} + N_k N_{k-1}t_{\text{fetch}} + N_k t_{\text{storage}} \quad (5-7)$$

(for a particular processor assigned to a layer of weights)

In Figure 5-3 processor k_1 has been assigned to the largest layer of weights and consequently has the largest fetch, execution, and storage times (depicted linearly). Processor k_1 is therefore termed the major processor and all other processors are referred to as being minor. Notice that although k_2 and k_3 show longer execution times, in reality their actual computation time would be much shorter. The purpose of extending the execution blocks is to show that due to the previously mentioned constraint, the computation portion of each of these cycles do not have to reside strictly inside the k_1 execution time window. The minor



processors execution time can be extended beyond the major processor's execution window and thus overlap the major processor's storage and fetch window.

As mentioned before, the computation time for the major processor must exceed the sum of all other minor processor's fetch/storage times. Whether the storage block is placed in front of the fetch block (as shown) or behind it, processors k_2 and k_3 can use the time in-between cycles for computing. What is important and necessary is that all processors produce outputs which are required by the next layer or final output in every cycle. Hence all processors must store all their previous state's computations in the present cycle and also fetch the new data during that cycle.

5.3.2 Pipelining and Local Memory

Referring back to Figure 5-4, shared local memory consists of blocks of RAM which are accessed by either one or two processors. When the network is in retrieval mode this topology has several advantages over the globally shared memory configuration. The first being an ability now to perform either concurrent fetches or concurrent stores but not both types of memory transactions at the same time. Because the network is flowing in the forward (right) direction only, all layer processors can store their results concurrently to the right. Similarly all processors can fetch data concurrently from the left. This alleviates the major processor execution/minor processors memory transaction constraint previously discussed. However this new architecture does not improve the overall pipeline throughput given by equation (5-5) or the speed-up factor.

This implementation does allow for vertical overlap both in the storage times for each k_{1-3} processor and the fetch portions for each cycle. Thus all minor processors must only have shorter individual storage, fetch and execution times as compared to the major processor which is a much more feasible system.

5.3.3 Applying Pipelining to Retrieval (Summary)

For the optimum case where a network's layers of weights are all relatively the same size (i.e. $W_1 \approx W_2 \approx \dots W_{K_L}$), the speed improvement (S_f) from pipelining with general purpose layer processors is approximately K_L (the number of layers in the network). In the case where a network has a single layer weights which dwarfs all other weight layers, $S_f \approx 1$. Thus pipelining is most effective for networks with large numbers of weight layers and optimal when the layers are similar in size (PE layers are constant in size).

In a pipelined system with non-concurrent read/write global memory, all minor processors must be able to perform their fetch/store operations during the execution block of the major processor. This is highly unrealistic for networks with constant weight layers or

networks with numerous weight layers. The memory transaction times for most DSPs are slower than their multiply/accumulate instruction cycles [41] [42]. Hence the throughput period described by equation (5-5) will now have to be widened to include the extra time lost by the minor processor's non-overlapping fetch/ storage operation with respect to the major processor execution block. These are the extra minor processor memory cycles that don't fall within the major processor's execution block.

If a system has been designed to accommodate overlapping fetch, execution and storage cycles between the layer processors, the next main bottleneck is the total input and weight fetch time. Due to fast microprocessor multiply and accumulate instructions, the largest portion of time in an activation computation cycle is not spent in calculation but is in data communication. The throughput period of the pipeline therefore approaches $W_{k\text{-fetch}}$ as the networks become increasingly large. Where W_k is the largest layer of weights in the network.

In summary, by applying ATPD to the multi-layer perceptron retrieval mode process and constraining it with various pipeline architectures, we can observe how the computation time improves or degrades. Certain pipelining architectures were found to create further process constraints (i.e. data bottlenecks) which will lower the initial computational bandwidth improvement.

5.4 ATPD Analysis of Backpropagation

Algorithmic Timing Parameter Decomposition will now be applied to the Backpropagation process described in Chapter 2 in equations (2-1) through (2-7). The initial architecture is assumed to be a serial computer. The reader should also refer back to Figure 2-4 for a block diagram representation of the flow. For ease of viewing purposes, both the equations governing backpropagation and the corresponding ATPD computation time

derivation will be shown together. See equation (5-8). The process is also broken into phases for further clarity.

1st Phase - Retrieval

$$y_i^k = \sigma \left(\sum_{j=1}^{N_{k-1}} (w_{ij}^k y_j^{k-1}) \right)$$

Retrieval Computation Time (RCT)

$$RCT = \sum_{k=1}^{K_L} N_k N_{k-1} t_f + N_k N_{k-1} (t_f + t_{ma}) + N_k t_{nlt} + N_k t_s \quad (5-8)$$

The primitives t_f and t_s are equivalent to the incremental amount of time required by the serial processor to either fetch or store one item from/to local memory. Similarly, t_{ma} is the unit time for a multiply-accumulate operation and t_{nlt} is the nonlinear transform elementary time unit. With these primitive definitions, the first term in (5-8) corresponds to fetching the inputs required for a layer of net computations in (2-1). The second term in the summation of (5-8) equates to the time that it takes to fetch the weights and perform the necessary dot product. The third term is the nonlinear transform of the nets (unit time) and the last term in (5-8) represents storing the resultant activations to local memory. N_{k-1} , once again, is the number of inputs or PE activations from the previous layer while N_k is the number of activations being computed in the k (right most) layer. RCT is therefore the total amount of time required to produce (retrieve) one output vector during retrieval.

In order to maximize space, the ATPD computation time equations will be explained via abbreviated descriptions rather than space consuming sentences. See (5-9). The new primitive t_{sub} is equivalent to the amount of time required to subtract two values in the serial processor's arithmetic logic unit (ALU). The third phase, the amount of time to propagate the output error through the PE layer is given by (5-10). (5-10) contains a

2nd Phase

Output Error Generation

$$e_i^k = d_i^k - y_i^k$$

Output Layer Error Computation Time (OLEC)

$$OLEC = N_{K_l} t_f + N_{K_l} t_f + N_{K_l} t_{sub} + N_{K_l} t_s \quad (5-9)$$

3rd Phase

Backpropagation of the Error
Through the PEs

$$\sigma'(u_i^k) e_i^k = y_i^k (1 - y_i^k) e_i^k$$

Back Err through PEs Computation Time (BEPE)

$$BEPE = \sum_{k=1}^{K_L} N_k t_f + N_k t_{nlt d} + N_k t_f + N_k t_{mul} + N_k t_s \quad (5-10)$$

new primitive $t_{nlt d}$ which is the unit time to compute the nonlinear transform derivative.

This primitive is dependent on the type of nonlinear transform used (i.e. sigmoid function or hyperbolic tangent) and therefore is shown as a generalized parameter in (5-10). The summation is from 1 to K_L even though the propagation is now in the reverse direction. While this is equivalent to summing from K_L to 1, it is written in a more recognizable form.

The 4th phase is shown next in (5-11) and is equivalent to backpropagation the error terms generated by (5-10) through the weights. Both the 3rd and 4th phases will need to be computed more than once in the case where K_L (the number of network layers) is

greater than 1. Because equations (5-10) and (5-11) may be computed several times, their time formulas contain k and $k-1$ subscripts rather than the K_L as in (5-9).

4th Phase

*Backpropagation of the Error
Through the Weights*

$$e_j^{k-1} = \sum_{i=1}^{N_k} w_{ij}^k \sigma'(u_i^k) e_i^k$$

Back Err through Wts Computation Time (BEWTS)

$$BEWTS = \sum_{k=2}^{K_L} N_k t_f + N_k N_{k-1} (t_f + t_{ma}) + N_{k-1} t_s \quad (5-11)$$

fetch $\sigma'(u_i)e_i$ terms $\nearrow$ $\nearrow$ $\nearrow$ $\nearrow$ store hidden err
 fetch wts., mult. and accumulate

5th Phase

Delta Wt. Generation

$$\Delta w_{ij}^k = \mu \sigma'(u_i^k) e_i^k y_j^{k-1}$$

Delta Wt. Computation Time (DWTS)

$$DWTS = \sum_{k=1}^{K_L} N_k t_f + N_k t_{mul} + N_{k-1} t_f + N_k N_{k-1} t_{mul} + N_k N_{k-1} t_s \quad (5-12)$$

fetch $\sigma'(u_i)e_i$ terms $\nearrow$ $\nearrow$ $\nearrow$ $\nearrow$ $\nearrow$ store Δw
 mult. by μ fetch $k-1$ activations mult. act. and err.

The 5th phase, the delta weight computation, is now given by (5-12). The last phase, the updating of the weights is shown in (5-13). A new primitive t_{add} has been generated for the weight update computation time. This elementary unit time equates to the amount of time required to add two values in the serial processor's ALU.

Having tediously compiled all the computation time equations found in (5-8) through (5-13), the formulas can now be added to yield the amount of time required to

6th Phase

Weight Update

$$w_{new}^k = w_{old}^k + \Delta w_{ij}^k$$

Updating the Wts. Computation Time (UWTS)

$$UWTS = \sum_{k=1}^{K_L} N_k N_{k-1} (2) t_f + N_k N_{k-1} t_{add} + N_k N_{k-1} t_s \quad (5-13)$$

↑ ↑ ↑
fetch old wts and Δwts perform update store new wts

perform one learning iteration. See equation (5-14). SCBP is the sum of all the times given by Phases 1-6 in equations (5-8) through (5-13). It is equivalent to the amount of time required to forward propagate an input through the net, compute the output error between the desired and output activation, backpropagate the error to the hidden layer PEs and eventually update the weights. One divided by SCBP is the backpropagation *computational bandwidth* of a particular system. This corresponds to the number of complete weight updates than can be performed per second.

Serial Computer Backpropagation Computation Time

$$SCBP = RCT + OLEC + BEPE + BEWTS + DWTS + UWTS \quad (5-14)$$

Equations (5-8) through (5-14) can also be used to identify exactly where weaknesses (time consuming operations) are in the system and test different ideas for reducing them. This process will be exemplified in the next section and performed numerous times throughout the remaining portion of this dissertation.

5.5 Adapting the ATPD Backpropagation Equations to Different Architectures

Equations (5-8) through (5-13) are a worse case estimate for computing the backpropagation algorithm on a serial machine. For example, nearly all present day processors (special purpose, DSP or general purpose) contain some type of internal memory for

maintaining intermediate results. If we assume that the serial processor used in section 5.4 has this feature, the following changes can be made to computation time equations (5-8) through (5-13). During retrieval (Phase I), the N_{k-1} inputs are read in once and maintained locally. Additional fetches of these values are performed internally and therefore the fetch time is lumped into the multiply-accumulate command. See the first term in equations (5-8) and (5-15). The only remaining N^2 term in (5-15) is now the term related to fetching the weights and performing the multiply-accumulate operation. This is a substantial change in the Retrieval Computation Time formula. Similarly alterations can be made for the output layer error computation, backpropagation of the error through the PE and backing the error terms through the weights. Intermediate error values do not need to be sent to external memory if they are required by the following computation or phase. However these are N size reductions and therefore are not as significant as an N^2 term reduction (i.e. as shown in retrieval). For large networks ($N \gg 100$ PEs per layer) these timing terms can be ignored when compared to the N^2 term. Thus the remaining computation equations (3-9)-(3-13) will be left intact. They will change however if an architecture has a drastic modification in hardware when compared to the basic serial computer. This will be shown for a systolic array processing element that contains additional internal parallelism exists.

New Retrieval Computation Time (RCT)

$$RCT = \sum_{k=1}^{K_L} N_{k-1}t_f + N_k N_{k-1} (t_f + t_{ma}) + N_k t_{nlt} + N_k t_s \quad (5-15)$$

5.6 ATPD Analysis of Neuro Turbo's DSP System and Kung's Systolic Arrays

Rather than develop a model (algorithm and architecture) from scratch that partitions the PEs in a given layer to several processors, Neuro Turbo and Kung's systolic array processors were analyzed as a point of reference. Both systems apply parallelism inside a

particular layer of PEs/weights as opposed to assigning the system processors to several PEs in different layers.

Neuro Turbo has an architecture that follows the locally shared memory structure shown in Figure 5-2. However Neuro Turbo forms a ring array where the last process communicates with the memory block at the front of the first processor. The systolic array also can be classified as a locally shared memory system where the memory and processor blocks are integrated as one array processing unit. See Figures 3-20 and 5-2.

5.6.1 Neuro Turbo's Computational Time Formulas

In the Neuro Turbo DSP ring all computations (retrieval and learning operations) can be performed in parallel locally by the DSPs. Therefore the serial computation time corresponding to equations (5-9) through (5-15), derived to estimate the serial processor's learning computation time, can be divided by the number of DSPs (4) in Neuro Turbo. However efficiency will be lost due to the additional time generated by storing/reading partial nets to the dual port memories. This is time lost transmitting the K_{xxn} terms (partial nets found in equations (3-6) through (3-9) around the ring. See Figures 3-8 and 3-9. The exact amount of this lost Dual Port Memory (DPM) communication time and the overall computation time for a single Neuro Turbo DSP will now be determined.

During retrieval, the K_{xxn} terms correspond to partitioning each complete net into 4 partial nets. For example, in equation (3-6), N_{0n} is broken into K_{00n} , K_{01n} , K_{02n} and K_{03n} . Thus the number of elements in K_{xxn} are equivalent to the number of nets represented by N_{0n} . Recalling that the nets computations were evenly distributed across all DSPs, N_{0n} is equivalent to $N_k/4$ in our notation (used in the computation time formulas, (5-9) through 5(15)). In the example where $N_k = 8$ PEs, N_{0n} represents two PEs (denoted N_{00} and N_{01}) and the corresponding partitioned nets (K_{xxn}) represent two values each. Thus in Fig. 3-8

the number of values written into a single dual port memory in the 1st cycle is $N_k/4$. Similarly in the 2nd cycle of Figure 3-8, $N_k/4$ values will be read out of DPM and again stored back into other DPMs. Note: The number of partial nets does not change. The DSPs are only adding values to the original K_{xxn} quantities. In addition, because all DSPs can read from their designated dual port memory in parallel, the time lost corresponds to retrieving $N_k/4$ partial nets.

The same holds true for the writing of $N_k/4$ partial nets. Knowing this and summing the number of read and writes to dual port memory, equation (5-16) is derived for Neuro Turbo retrieval. The first term is the serial computation time divided amongst the DSPs and

$$\begin{aligned} \text{Neuro Turbo} \\ \text{Retrieval} \\ \text{Computation Time} \end{aligned} = \frac{1}{4} [\text{RetrievalComputationTime}] + \quad (5-16)$$

$$\sum_{k=1}^{K_L} \frac{1}{4} [N_k (t_{sdpm})] + \frac{1}{4} [N_k (t_{fdpm} + t_{sdpm})] + \frac{1}{4} [N_k (t_{fdpm} + t_{sdpm})] + \frac{1}{4} [N_k (t_{fdpm})]$$

where;

- K_L = total number of layers in the network
- N_k = total number of inputs or PEs in a given k layer
- t_{fwm} = time required to fetch one datum from working memory
- t_{ma} = time to perform a multiply and accumulate
- $= t_{multiply} + t_{accumulate}$
- t_{nlt} = nonlinear transformation time
- t_{swm} = time to store one PE output in working memory
- t_{fdpm} = time required to fetch one datum from dual port memory
- t_{sdpm} = time to store one datum into dual port memory

the second term (summation) corresponds to the added time required to transmit partial nets around the ring. The primitives comprising (5-16) are now found to be dependent on the Neuro Turbo hardware. The efficiency of the system will be directly influenced by the

transfer rates between the working memory and dual port memory. However even if the dual port memory is several times slower than working memory, we still would expect the speed-up factor of this system to be nearly four for large networks ($N \gg 100$). This is due to the *local computation growing with respect to N^2 and global communication with N* .

Backpropagation in Neuro Turbo will be shown to produce very similar results to those found in (5-16). The serial computer backpropagation computation time described by equations (5-9) through (5-13) will be distributed across all four processors. This is deemed the local DSP computation time. Added to this is the time generated by transferring error terms around the ring for the backpropagation of the error through the weights operation. See Figures 3-11 and 3-12. The added dual port memory transfer time will turn out to be identical to the time required to transfer partial nets around the ring during retrieval. Specifically, this global communication time is added into equation (5-11), the backing of the error through the weights. See the 4th Phase in equation (5-17). Equation (5-17) is equivalent to the time required by Neuro Turbo to perform one learning iteration (a single update of all weights). All individual function equations comprising (5-17) are shown, including retrieval, for clarity purposes. The time corresponding to sending the error terms around the ring is identical to the partial sum communication time found in (5-16). The number of error terms passed to/from one dpm is exactly $N_k/4$ (see Figures 3-11 and 3-12 again). This is the same amount of values as in retrieval. In addition, the concurrent reading (writing) of values from (to) dual port memory by all processors also holds in this transfer operation. i.e. all DSPs can write (read) to (from) a dual port memory at one time.

$$\frac{\text{Neuro Turbo Backpropagation}}{\text{Computation Time}} = \quad (5-17)$$

1st Phase - Retrieval

$$\sum_{k=1}^{K_L} \frac{1}{4} [N_{k-1}(t_{fwm}) + N_k N_{k-1}(t_{fwm} + t_{ma}) + N_k(t_{nlt}) + N_k(t_{swm})] \\ + \sum_{k=1}^{K_L} \frac{3}{4} [N_k(t_{fdpm} + t_{sdpm})] \quad +$$

2nd Phase - Output Error

$$\frac{1}{4} [N_{K_L} t_{fwm} + N_{K_L} t_{fwm} + N_{K_L} t_{sub} + N_{K_L} t_{swm}] \quad +$$

3rd Phase - Backpropagation of the error through the PEs

$$\sum_{k=1}^{K_L} \frac{1}{4} [N_k t_{fwm} + N_k t_{nlt} + N_k t_{fwm} + N_k t_{mul} + N_k t_{swm}] \quad +$$

4th Phase - Backpropagation of the error through the weights

$$\sum_{k=2}^{K_L} \frac{1}{4} [N_k t_{fwm} + N_k N_{k-1}(t_{fwm} + t_{ma}) + N_k t_{swm}] \\ + \sum_{k=1}^{K_L} \frac{3}{4} [N_k(t_{fdpm} + t_{sdpm})] \quad +$$

5th and 6th Phases and - Dwt. generation and update

$$\sum_{k=1}^{K_L} \frac{1}{4} [N_k t_{fwm} + N_k t_{mul} + N_{k-1} t_{fwm} + N_k N_{k-1} t_{mul} + N_k N_{k-1} t_{swm}] \quad +$$

In the 4th phase of (5-17) it is also important to observe the starting index in each of the two summation terms. The local backpropagation of the error computation time summation begins with a '2' starting index because errors are not propagated back through the input layer weights. The DPM communication of errors summation however has a '1' as the starting index.

This is because the error terms must be passed around the ring for every layer of PEs.

The primitives required by (5-17) are the same as those shown in (5-16). All local communication is shown to be between a DSP and working memory (also referred to as local memory). Global communication is performed through the dual port memories using the fetch/store primitives t_{fdpm} and t_{sdpm} . In summary, Neuro Turbo should effectively reduce the serial backpropagation computation nearly four times for moderate ($N > 20$) and larger size networks. What this implies is that the global communication (DPM transfer times) found in phases I and IV of (5-17) is relatively insignificant when compared to the local processing time. Hence Neuro Turbo has a very efficient partitioning and scheduling scheme for paralyzing static neural networks.

5.5.2 Systolic Array Ring Computation Time

Kung's systolic arrays compute the retrieval operations in a slightly different manner. While the weights are maintained locally (and evenly distributed), the activations from the previous layer are passed around the ring. See Figure 3-15. After reviewing the literature on the actual array processor hardware it is evident that the weight and $k-1$ layer activation are shifted in concurrently into a particular array element [31]. With this in mind and returning to the serial computation time shown in equation (5-8), the first term in the summation, corresponding to the time needed to fetch the $k-1$ activations, can be removed.

In addition, the desired signal and error terms are stored in internal memory structures which can be accessed in parallel with external data. The activations are also main-

tained internally once a layer of retrieval has transpired. Hence for future operations, it will be assumed that they can be brought into the ALU in parallel with the weights and externally shifted values. This assumption requires separate data paths for internal memory, external weight memory and external shifted values. Since the systolic array processor is designed from scratch as a custom VLSI chip this should be feasible. The storing of results internally will also be assumed to be performed in parallel with the external shifting of input values (systolic beat). Hence all fetch and storage times for activations and errors will also be removed. See (5-18) and compare to (5-8).

$$\begin{array}{l} \text{Systolic Ring} \\ \text{Retrieval} \\ \text{Computation Time} \end{array} = \sum_{k=1}^{K_L} \frac{1}{K_{\min}} [N_k N_{k-1} (t_{fa} + t_{ma}) + N_k (t_{nlt})] \quad (5-18)$$

where;

- K_L = total number of layers in the network
- N_k = total number of inputs or PEs in a given k layer
- t_f = time required to shift in one datum into the array element
- t_{ma} = time to perform a multiply and accumulate
= $t_{\text{multiply}} + t_{\text{accumulate}}$
- t_{nlt} = nonlinear transformation time
- $K_{\min}$ = smallest PE layer in network also, no. of array processors

$K_{\min}$ corresponds to the minimum layer of PEs in the network and therefore is equivalent to the number of array elements in the ring. In the 4-12-8 neuron network example, $K_{\min}$ would equal 4. See Figures 3-16 and 3-17. The systolic cycle or heartbeat of the system is therefore equivalent to $t_f + t_{ma}$. Additional cycles are needed then to perform the nonlinear transform and store the resulting k layer activation. As a further check, it was mentioned in section 3.6.1 that 3 loop cycles or 12 shifts (heartbeats) were required to perform the multiply accumulate activity in Figure 3-16 and 24 shifts (6 loops times 4 shifts) for Figure 3-17. Applying (5-18), we calculate that it takes $N_k N_{k-1} / K_{\min}$ shifts to complete

the multiplication accumulation process (dot product). In Figures 3-16 and 3-17, $N_k = 12$ and 8, $N_{k-1} = 4$ and 12, respectively. Using the K_{min} value of 4, we obtain 12 and 24 shifts; which matches our previous results calculated in section 3.6.1.

Because of the internal architectural differences between the serial processor described in section 5.2 and the systolic array unit processor studied in sections 3.6.1 and 3.6.2, several alterations will need to be made to the backpropagation timing equations (5-8) through (5-13). For simplicity, the systolic array backpropagation time model will be shown first and then the alternations will be discussed phase by phase. See equation (5-19). Phase 1 remains the same as in (5-18). Activations are assumed to be shifted in parallel with the weights. As in retrieval the internal storing/fetching of activations and errors will also be assumed to be performed in parallel with the external shifting. Therefore phases 1 and 3-6 do not contain terms which correspond to fetching and storing activations or errors. Compare these phases with equations (5-8) and (5-10) through (5-13).

In phase 2, although the activations stored from the output layer are assumed to be local, there is no external shifting operation (pumping in a weight) and so the fetch time can not be neglected. (Recall that previously activation/error fetch and store times were removed due to the activity being performed in parallel with the external data shifting.) However the desired signal is assumed to be transferred in parallel with the activation to the ALU. Thus only one fetch term is required when compared to (5-9). This primitive is shown as t_{fint} which equates to a single *internal* fetch. This may be a faster time than t_f which is equivalent to shifting the weight into the systolic processor (external memory transfer).

The storage of the error term in phase 2 will be assumed to be performed in parallel with the subtraction. (i.e. subtract two values, store the result while the next subtraction is being performed.) This is another feature which should be designed into the VLSI chip.

$$\frac{\text{Systolic Array Backpropagation}}{\text{Computation Time}} = \quad (5-19)$$

1st Phase - Retrieval

$$\sum_{k=1}^{K_L} \frac{1}{K_{min}} [N_k N_{k-1} (t_f + t_{ma}) + N_k (t_{nlt})]$$

+

2nd Phase - Output Error

$$\frac{1}{K_{min}} [N_{K_L} t_{fint} + N_{K_L} t_{sub}] \quad +$$

3rd Phase - Backpropagation of the error through the PEs

$$\sum_{k=1}^{K_L} \frac{1}{K_{min}} [N_k t_{nlt} + N_k t_{mul}] \quad +$$

4th Phase - Backpropagation of the error through the weights

$$\sum_{k=2}^{K_L} \frac{1}{K_{min}} [N_k N_{k-1} (t_f + t_{ma})] \quad +$$

5th and 6th Phases and - Dwt. generation and update

$$\sum_{k=1}^{K_L} \frac{1}{K_{min}} [N_k N_{k-1} t_{mul}] \quad + \quad \sum_{k=1}^{K_L} \frac{1}{K_{min}} [N_k N_{k-1} (t_f + t_{add}) + N_k N_{k-1} t_s]$$

compute $\mu\sigma(u_i)e_i y_j^{k-1}$ terms first

shift in old weight and add μy_j^{k-1}

save new wts.

The weight update (phases 5 and 6) are shown as two simplified summations at the bottom of (5-19). Phase 5 and 6 are equivalent to the parallel shifting in of the weight (external memory), the $k-1$ activation (external) and the error term (internal memory). Refer back to Figure 3-23 for clarity. The multiplication by μ has already been performed and thus is removed from (5-12). The fetching of activations and error terms is over shadowed by the shifting in of the weights and thus their contributions in (5-12) are also removed. The Δw value is immediately being used to update the weight and hence the last storage term in (5-12) is also deleted. We are therefore left with only the N^2 multiplication term in (5-12) for phase 5. The second summation (phase 6) consists of shifting in the weights, adding the Δw value and storing the new weight. The fetching of the Δw values is not necessary (they were just computed and reside in the ALU) and thus this term is removed from (5-13).

In summary, the systolic array model has several simplifications when compared to equations (5-8) through (5-13). These simplifications however require parallel data paths (internal and external) and dedicated memories in the array processing unit. This complicates the design and increases the expense of the custom chip. However if one desires a high degree of parallelism (large K_{min} value) and is not concerned with economic cost this is a very efficient parallel realization. There are no added global communication costs; weights are shifted in at the same time as non-local values (i.e. activations or error sums).

This section also illustrates the value of breaking down retrieval and backpropagation into (5-8) through (5-13) via algorithmic timing parameter decomposition. The design of the internal systolic array processing unit directly affects which time terms can be removed from the summations in the timing equations. These equations should therefore be used as a template for what features (separate memories, data paths, and isolated data paths in the ALU) should be incorporated in the array processing element.

5.6 ATPD Analysis of the Focused Gamma Model

Applying ATPD to the focused gamma model is a relatively straight forward operation. From a computational point of view, the focused gamma model has many similarities when compared to the multi-layer perceptron. The main differences however are as follows: Before retrieval can proceed, systems processors must first compute the gamma kernel output values as governed by equation (4-1). Once these values are obtained, the net computations can proceed. This is very similar to the static net case with the only exception being that dot product contributions will need to be obtained from the gamma weight layers. See equation (4-2).

During backpropagation, two differences arise in that extra time is required for updating the gamma weights and adapting the gamma parameter. Updating the gamma weights does not require any additional backpropagation of the error. However in order to update the gamma parameter, error terms must be backpropagated through the gamma weights to obtain final error sums at each kernel output. These additional effects will now be quantified.

Rather than starting from scratch in developing a computation time model for the focused gamma net, the serial computer backpropagation computation time, SCBP, equation (5-14) will be adapted to reflect the additional computation times mentioned above. This is termed SCBP_FGM. Starting with retrieval, equation (5-15) is expanded to first include gamma kernel output generation and then forward propagation of the input and gamma output vectors. See equation (5-20). N_0 is the number of inputs in the focused gamma net and N_1 is the number of PEs in the first layer. In (5-20), the first main term corresponds to computing the gamma kernel outputs. This is broken down into first fetching the gamma parameter and then subtracting it from one. This operation is done once for every gamma structure and thus is N_0 in size. Equation (4-1) should be revisited if this operation is un-

Focused Gamma Retrieval Computation Time (FGRCT)

$$\begin{aligned}
 FGRCT = & N_0 ((t_f + t_{sub}) + G(2t_f + 2t_{ma} + t_s)) \\
 & + G(N_0 t_f + N_0 N_1 (t_f + t_{ma})) \\
 & + \sum_{k=1}^{K_L} N_{k-1} t_f + N_k N_{k-1} (t_f + t_{ma}) + N_k t_{nlt} + N_k t_s
 \end{aligned} \tag{5-20}$$

clear. The second portion of this main term consists of fetching the previous gamma kernel outputs (saved from the last state) and multiplying them by $1-\gamma$ or γ according to Figure 4-3. The new kernel output is then stored back to memory.

The next main term equates to fetching the kernel outputs, gamma weights and obtaining the net shown in (4-3). G is the number of gamma kernels present in the gamma structure. Once a net term has been obtained it is not stored to memory, instead it is left in the accumulator for addition to the dot product sum through the $g=0$ layer of weights. The $g=0$ net computation time and additional $k>1$ retrieval times are given by the last major term in (5-20).

Moving on to backpropagation, the output error time computation remains the same as that shown in (5-9). The backpropagation of the error through the PEs also does not change from equation (5-10). The backpropagation of the error through weights, (5-11), however will now have to change to reflect the backpropagation of the error through the gamma weights. See equation (5-21). The summation corresponds to backing the error through the weights as in the static model and the time required to back the error through the $g=0$ first layer weights. The second term therefore corresponds to backing the error through the gamma weights. G is the number of kernels in the gamma structure. N_0 and N_1 are again the number of inputs and PEs in the first layer.

The next phase consists of computing the delta weight terms for all the weights in the focused gamma network. See (5-22). The first term consists of updating all the weights that are common to both the multi-layer perceptron and focused gamma net. The second term is the contribution of time associated with computing deltas for the gamma weights. The terms which comprise it are explained as those shown under (5-12).

Focused Gamma - Backpropagation of Err through Wts Computation Time (FGBEWTs)

$$FGBEWTs = \sum_{k=1}^{K_L} N_k t_f + N_k N_{k-1} (t_f + t_{ma}) + N_{k-1} t_s \quad (5-21)$$

$$+ G (N_1 t_f + N_0 N_1 (t_f + t_{ma}) + N_0 t_s)$$

Focused Gamma Delta Wt. Computation Time (FGDWTS)

$$FGDWTS = \sum_{k=1}^{K_L} N_k t_f + N_k t_{mul} + N_{k-1} t_f + N_k N_{k-1} t_{mul} + N_k N_{k-1} t_s \quad (5-22)$$

$$+ G (N_1 t_f + N_1 t_{mul} + N_1 t_f + N_0 N_1 t_{mul} + N_0 N_1 t_s)$$

Similarly, the weight update equation can be expanded from (5-13) to that shown in (5-23). The first summation consists of updating all the weights common to the MLP and

Focused Gamma Updating the Weights Computation Time (FGUWTS)

$$FGUWTS = \sum_{k=1}^{K_L} N_k N_{k-1} (2) t_f + N_k N_{k-1} t_{add} + N_k N_{k-1} t_s \quad (5-23)$$

$$+ G (N_1 N_0 (2) t_f + N_1 N_0 t_{add} + N_1 N_0 t_s)$$

focused gamma net. See equation (5-13). The new term tacked on to (5-13) in (5-23) represents the amount of time required to update the gamma wts. While no other computation

time equations are needed for the MLP, one last phase is required for the focused gamma net. This is the time required to compute the delta gamma parameters and update the old gamma parameters. The delta was given in (4-8) and the update was said to be the same as that in weight updating. For brevity both computation times are grouped and presented in (5-24). The first term consists of fetching the two kernel outputs (saved from the last state, fetching the associated error term, subtracting the kernel values and multiplying them by the error. See (4-7) for review. Once these sums have been obtained, they are multiplied by μ and saved. This is represented by the second term in (5-24). The last term corresponds to updating the gamma parameters and is similar to updating the weights in (5-23). However because there is only one gamma parameter per kernel structure or net input, the quantity is N_0 in dimension.

Focused Gamma - Delta and Gamma Parameter Update Computation Time (FGDPU)

$$FGDPU = GN_0(3t_f + t_{sub} + t_{ma}) + N_0(t_{ma} + t_s) + N_0(t_f + t_{add} + t_s) \quad (5-24)$$

Summing all of these terms we obtain the Serial Computer Backpropagation Computation Time for the focused gamma model. See (5-25). These results will be used later in Chapter 8 for predicting execution times for focused gamma nets.

Focused Gamma Serial Computer Backpropagation Computation Time (FGSCBP)

$$FGSCBP = FGRCT + OLEC + BEPE + FGBEWS + FGDWTS + FGUWTS + FGDPU \quad (5-25)$$

5.7 Conclusions

Although Kung's ring arrays have excellent computational bandwidths, they also have several other inherent negative aspects. The first being that the array processing element is a relatively complex unit and therefore must be fabricated using VLSI technologies.

No “off the shelf” components can be used to implement the basic element without a significant sacrifice in performance and cost. Another element against the choice of using the array ring is its lack of flexibility in the type of task it performs. The ring has been designed strictly for static multi-layer perceptron neural computing; where as, Neuro Turbo can be used in many preprocessing applications (i.e. filtering, FFTs, etc...) because of its general purpose DSPs and architecture. A further problem is that the ring size is dependent on the minimum layer of PEs in the static net. All other layers in the net are expected to be integer multiples of this value or dummy PE padding is required for partitioning the layer onto the ring. This creates poor results for networks that have PE layers which vary greatly in size. The degree of parallelism is dependent on the minimum layer; this may be inadequate for other significantly larger layers in the network.

While Neuro Turbo offers the widest variety of signal processing techniques, the additional time created by the transferral of the partitioned dot-product sums and errors between DPMs slightly degrades the parallel efficiency of the system. This suggests that the speed-up factor will be nearly equal to the number of DSPs ($P=4$). This may prove to be unacceptable for many applications. Especially for smaller networks, where global communication is a more significant time contributing factor.

Because of these disadvantages, a third type of system is proposed to provide both computational flexibility and efficient parallelism. This system is called the CNEL Machine and is presented in Chapter 6.

CHAPTER 6 MULTIPLE DSP ARCHITECTURES

6.1 Introduction

After reviewing several neural technologies (e.g., commercially available analog hardware, the Connection Machine, systolic arrays, DSP and/or transputer based systems) and opening discussions with the neural network researchers from my lab, the goals for our system began to emerge. The Computational NeuroEngineering Lab (CNEL) members expressed desire for a system that would be computationally flexible, inexpensive, fast and readily available to a group of users.

The need for computational flexibility arose out of both the pre-processing requirements of certain applications and the wide variety of research being performed in the lab. A brief example will now be presented. Three CNEL students working on masters degrees were involved in sampling (recording) time domain signals, converting these files to the frequency domain and then using the spectra to train a network. Two of the students were working on musical pitch detection algorithms and the third was involved in isolated word recognition. All three students used a NeXT Machine as their computational platform. While the sampling and conversion to the frequency domain was not computationally intensive, the network training took several (14+) hours to converge. These users therefore would have to wait overnight to get results back on testing new methods or ideas. After talking to them, we decided that what they needed was a system which would speed up the training at least 50 times (when compared to a NeXT). In addition, we wanted the system to have the ability to receive time domain data (files) and convert it quickly to spectral

Table 6-1. Benchmark Neural Program Execution Times (min:secs) and Compiler

<u>NeXT</u> <u>(25 MHz)</u>	<u>SUN</u> <u>SPARC 2</u>	<u>SUN (50MHz)</u> <u>SPARC 10</u>	<u>PC486DX</u> <u>(33 MHz)</u>	<u>PC486DX2</u> <u>(66 MHz)</u>
3.08	3:00	1:10	6:15	3:10
GNU C -O4	SUN C version 2.01 -O4		Visual C++ 1.0 (full optimization)	

information. With this feature, the system could be connected to a PC or other inexpensive front-end platforms.

Because of our limited budget, it was also deemed that whatever solution was proposed, it should cost in the low several thousand dollars range. Ideally, we were looking for a system which could speed-up our jobs at least 50-100 times for less than \$5,000. If we could achieve this performance/cost ratio, we would possibly buy or build several of these systems for connectivity to our PCs and/or NeXT stations. Having several of these systems (each with a 50+ speed-up factor) would allow many users to obtain results in a few hours rather than overnight.

6.2 Narrowing Down the Spectrum of Available Neural Technologies

A work station (serial machine) was considered first as a possible platform for accelerated processing. We limited our selection to those in the \$10K range such as the cheaper SUN, HP, or IBM RISC systems. However after benchmarking a general purpose neural network package (written in C) on a few machines, it became apparent that a single work station would be too slow. See Table 6-1. The benchmark program was compiled on each individual machine and timed for execution. The compiler and optimization level employed is shown under each machine's time in Table 6-1. The program functions as a

time delay neural net (TDNN) and is used for signal prediction. The TDNN consists of 10 tapped delay inputs, 20 hidden layer PEs, and one output PE. The training input set and desired signal consisted of three thousand points each and weights were updated for each new input point. All execution times shown in Table 6-1 are in minutes:seconds and should be cross referenced with the NeXT Machine's execution time (3:08). The PCs were tested to yield information as to where they stand when compared to SUN's SPARC machines. From the results shown in table 1, it can be seen that our speed-up goal (50+ times the NeXT Machine) cannot be delivered unless groups (a network cluster) of machines are used. This will be an important consideration explored in the last chapter. Hewlett Packard and IBM latest work stations are expected to show similar results (near the same order of magnitude speed-up) as the SUN SPARC 10.

Because a serial machine couldn't deliver the cost/speed-up performance goal previously described in section 6.1, parallel machines were then considered. Returning to Figure 3-1, all non-digital systems were passed over due to their high cost, limited precision and instruction inflexibility. Moving into the digital domain (fine grain parallel machines), an account was offered to us from Florida State University's super computing facility for use of the Connection Machine (CM). This machine however was not experimented with for the following reasons. The first is that the manual which supported this system was difficult to follow. The author had reservations as to how well the lab would be able to grasp CM's architecture and manipulate CM programs written by the author. In addition, the paper discussing network partitioning on CM was also difficult to follow [3]. Thus all reference material (for lab members) on how to use the CM and partition nets across it would have to developed solely by the author. Another area of concern was the general availability of the machine. Would the machine always be available for all CNE Lab members and at any given time? This is highly unlikely. Finally one last negative

factor on developing code on the CM is the cost of a single machine and its portability. A Connection Machine costs several million dollars and must reside in a cool fixed location. This would limit the use of any developed software/partitioning schemes from individuals who don't have access to this parallel computer (e.g., small colleges, universities, companies). In summary, because of these reasons, the Connection Machine (or any other fine grain massively parallel system) was not chosen as the hardware research platform for accelerated neural processing.

Array processors were also not chosen because of cost and flexibility. The cost of designing a custom VLSI chip and mean time before a system would be available was deemed to be beyond our budget and too lengthy. The range of processing function on a array system was also felt to be too limited. See section 5.6 for a more detailed description of the negative aspects of using such a system.

Thus in order to meet the goals laid out in the previous section, coarse grain parallel processing was selected as the area for further study. This narrows down the choices to a system comprised of several (e.g., 2-20) digital signal processors (DSPs) or transputers. DSPs were chosen over transputers for the following reasons. The technology used to implement the system should be driven by or reside in the general personal computer (PC) market. e.g., PC memory, processors, PALs, and other currently commercially available hardware. It was felt that DSPs (Texas Instruments and Motorola) are more directly tied to this market than transputers. (The availability of DSP boards is greater for PCs as compared to transputer boards.) In addition, the currently available software (compilers, debug tools, etc...) to develop a system based on DSPs is extensive and available as shareware in many cases. Because of this available supporting software, processor functional flexibility and declining processor prices, DSP based systems were selected as the architectural platform for future study.

6.3 Three Basic DSP Architectures

Three elementary DSP architectures will now be analyzed using algorithmic timing parameter decomposition. The serial machine backpropagation equations will be adapted for each of the three specific architectures. The goal of this work is to determine which architecture is best suited for neural processing (economic cost vs. performance). See Figures 6-1 through 6-3. Figure 6-1 illustrates an architecture containing a single global memory bus. The shared static RAM is assumed to be either read only or write only by a

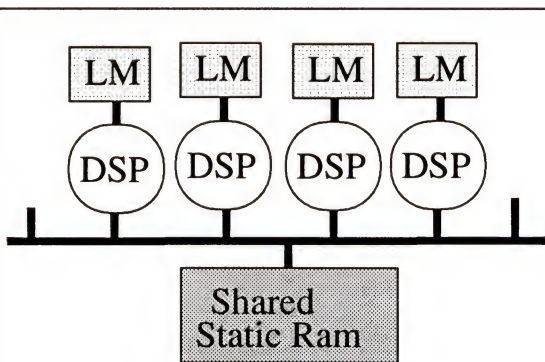


Figure 6-1. Common Bus Architecture

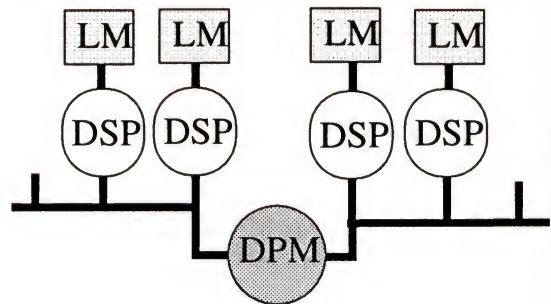


Figure 6-2. Dual Bus Architecture

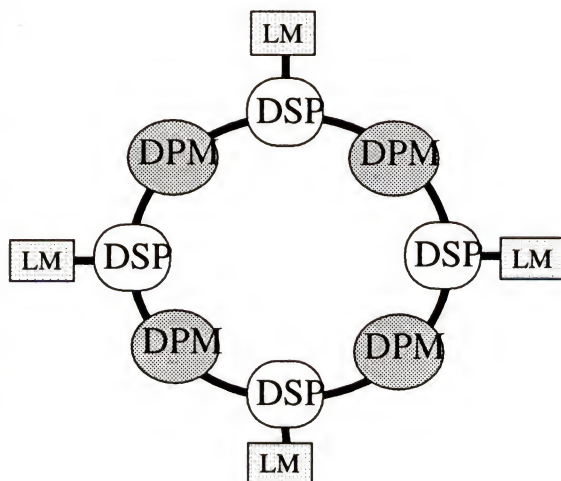


Figure 6-3. Ring Configuration

Key

— = general memory bus

LM = Local Memory

DSP = Digital Signal Processor

DPM = Dual Port Memory

single DSP. All DSPs therefore must share the global memory block. This block will be used to transfer data required by all processors (globally transmitted results). Local memory will be used to store weights and is normally much larger than the shared (global) memory block. Figure 6-2 replaces the single machine read/write memory found in Figure 6-1 with a dual port memory (DPM). Dual port memory can be written to or read from simultaneously by two processors as long as they reside on each side of the DPM. The single global memory bus found in Figure 6-1 can be split into two buses as shown by Figure 6-2. Figure 6-3 expands the dual bus further by placing a DPM between each DSP. (e.g., DSPs share a DPM between them.) This is the same configuration used in Neuro Turbo.

Having fixed these architectural configurations, a partitioning scheme must be selected to complete the diagram found in Figure 1-1. One goal of the partitioning scheme should be that it distribute network weights, inputs, desired signals and activation calculations evenly across all system processors. Another goal is that the global communication be minimal when compared to the local processing. In sections 3.5.1 through 3.5.4, it was found that Neuro Turbo's partitioning scheme met these requirements and thus will be used for all three experimental architectures. This partition scheme distributes all the weights to the system processors according to backpropagation alignment and thus is referred to as partitioning according to backpropagation alignment. Inputs and desired signals are evenly divided across all processors. In addition, PE activation calculations (one layer at a time) are also divided amongst the system processors evenly. Therefore if the number of processors in the system is P , then the number of activations assigned to a particular processor is N_k/P . The first processor in the system would therefore compute the top N_k/P PEs in a k layer of PEs shown in Figure 2-3. The second processor would compute the next N_k/P activations and so forth until all activations are distributed across P processors. For further detail review section 3.5.1.

6.4 ATPD Analysis of Three Basic DSP Architectures

Beginning with the common bus architecture, equation (6-1) yields the time required to propagate a single input vector through the network (multi-layer perceptron retrieval). It should be mentioned that the three architectures were critiqued for their efficiency in partitioning the multi-layer perceptron rather than the focused gamma model because this was considered a worse case. As explained in section 4.5, the focused gamma model was shown to be more efficient an algorithm to partition across a group of processors when compared to the MLP. While the focused gamma net's global communication time is equivalent to the global communication in the MLP, the local processing time is much greater than the MLP's. Thus the degree of parallelism should be higher on a multi-processor system for the focused gamma net. Because of this, architectural modeling was based on partitioning the MLP with the assumption that the focused gamma model would out perform the static neural network in terms of parallel efficiency.

Common Bus Retrieval Time (CBRT)

$$CBRT = \frac{1}{P}RCT + (P - 1) \sum_{k=1}^{K_L} N_k (t_f + t_s) \quad (6-1)$$

In equation (6-1) the first term, $(1/P)RCT$, is the retrieval computation time for a serial machine found in (5-15) divided by the number of processors in the system. The second term is the amount to time required to send partial nets between processors for retrieval. Recalling retrieval under the backpropagation alignment partitioning scheme, each processor produces a partial net for all N_k PEs. Once the partial nets have been generated, each processor keeps the PE partial nets assigned to it and transmits the partial nets required by the other processors. For example, if $P=4$ and $N_k=20$ PEs, each processor will first compute 20 partial nets and maintain locally the 5 partial nets assigned to it. The other

15 partial nets, it has generated, are placed into memory so that the other processors can obtain them. Once in global memory, each processor then reads (from global memory) the partial net values generated by the other processors (3 times 5) which correspond to the PEs they have been assigned. They then recombine these partial nets, 4 values per PE (one partial net from each processor), such that 5 completed nets are obtained for their 5 assigned PEs. Upon generating the 5 assigned nets, the values are input into a non-linear transform look-up table to produce the 5 activation values. This process then repeats for additional layers.

In general, the number of assigned PE activation calculations per processor is N_k/P . The number of partial nets placed written into global memory by a single processor is therefore $N_k - [(N_k/P)]$ which can be re-written as $[(P-1)/P]N_k$. The number of values which must be read from global memory per single processor is $(P-1)[(1/P)(N_k)]$. Because this memory can only be accessed by one processor at a time and concurrent writing/reading is not permitted, the total number of reads or writes is $P(P-1)[(1/P)(N_k)]$ which simplifies to $(P-1)(N_k)$ as found in equation (6-1). t_s and t_f are the primitives associated with storing (writing) and fetching (reading) one value to/from global memory. The time required to re-combine partial nets is assumed to be negligible when compared to RCT/P .

A similar analysis can be applied to the dual bus architecture shown in Figure 6-2. The dual bus retrieval time (DBRT) is given in (6-2). The only change from (6-1) is that

Dual Bus Retrieval Time (DBRT)

$$DBRT = \frac{1}{P}RCT + \frac{(P-1)}{2} \sum_{k=1}^{K_L} N_k (t_{fdpm} + t_{sdpm}) \quad (6-2)$$

now two processors (one on each side of the dual port memory) can access memory simultaneously. The global access time is therefore effectively divided by 2. The primitives have

also been altered to reflect fetching/storing from/to a dual port memory rather than conventional static RAM.

Moving to the ring configuration, Figure 6-3, the retrieval time can be estimated as in equation (6-3). In the ring all processors can write simultaneously to their neighboring DPMs. They can also read simultaneously and thus the global access time is divided by P rather than 2 in equation (6-2). The scheduling of the partial net calculations is slightly more complicated for the ring than the previous two architectures. See Neuro Turbos retrieval process in section 4.5.1. Specific partial nets are computed in a particular order. Also note that if P is made equal to 4 in (6-3) the resulting retrieval computation time matches that in equation (5-15), Neuro Turbos retrieval computation time.

Ring Configuration Retrieval Time (RCRT)

$$RCRT = \frac{1}{P}RCT + \frac{(P-1)}{P} \sum_{k=1}^{K_L} N_k (t_{fdpm} + t_{sdpm}) \quad (6-3)$$

Backpropagation will now be analyzed for each of the three architectures. Beginning with the common bus system, equation (6-4) is equivalent to the amount of time required to perform one learning iteration (single update of all network weights). The first

Common Bus Backpropagation Time (CBBT)

$$CBBT = \frac{1}{P} (SCBP) + (P-1) \sum_{k=1}^{K_L} N_k (t_f + t_s) + \sum_{k=1}^{K_L} N_k ((P-1) t_f + t_s) \quad (6-4)$$

term in (6-4) is the local serial computer backpropagation time (SCBP) given by (5-14) now divided across the P system processors. The second term in (6-4) corresponds to the global communication time associated with retrieval. See equation (6-1). The last term in (6-4) is equal to the time required to exchange error terms between all processors. This is a

new term and is generated by the following action. Upon completing retrieval, all processors begin computing output errors for each output layer PE assigned to it. These errors are then propagated back through the output PEs. Once complete, the local error terms are required by all processors for updating the local weights and/or backpropagation through the output weights (hidden layer error generation). Recalling the backpropagation alignment weight storage, this necessitates the need for all processors to have access to all layer errors. Because of this, each processor must send N_k/P errors and read $N_k - (N_k/P)$ errors from global memory. Summing this time for all processors, multiplying by P , we obtain the last term in (6-4).

Replacing the conventional global SRAM with a dual port memory yields equation (6-5). The first term again corresponds to the serial computer backpropagation time divided by the system processors and the second term is the dual bus retrieval time. The third time quantity is the DPM version of the last term in (6-4). Because two processors can simultaneously access memory, the global error transmission term in (6-4) is halved for (6-5).

Dual Bus Backpropagation Time (DBBT)

$$DBBT = \frac{1}{P} (SCBP) + \frac{(P-1)}{2} \sum_{k=1}^{K_L} N_k (t_f + t_s) + \frac{1}{2} \sum_{k=1}^{K_L} N_k ((P-1) t_f + t_s) \quad (6-5)$$

Adding a DPM between each processor produces the backpropagation time estimate found in (6-6). The first two terms are generated in a similar manner as in (6-5).

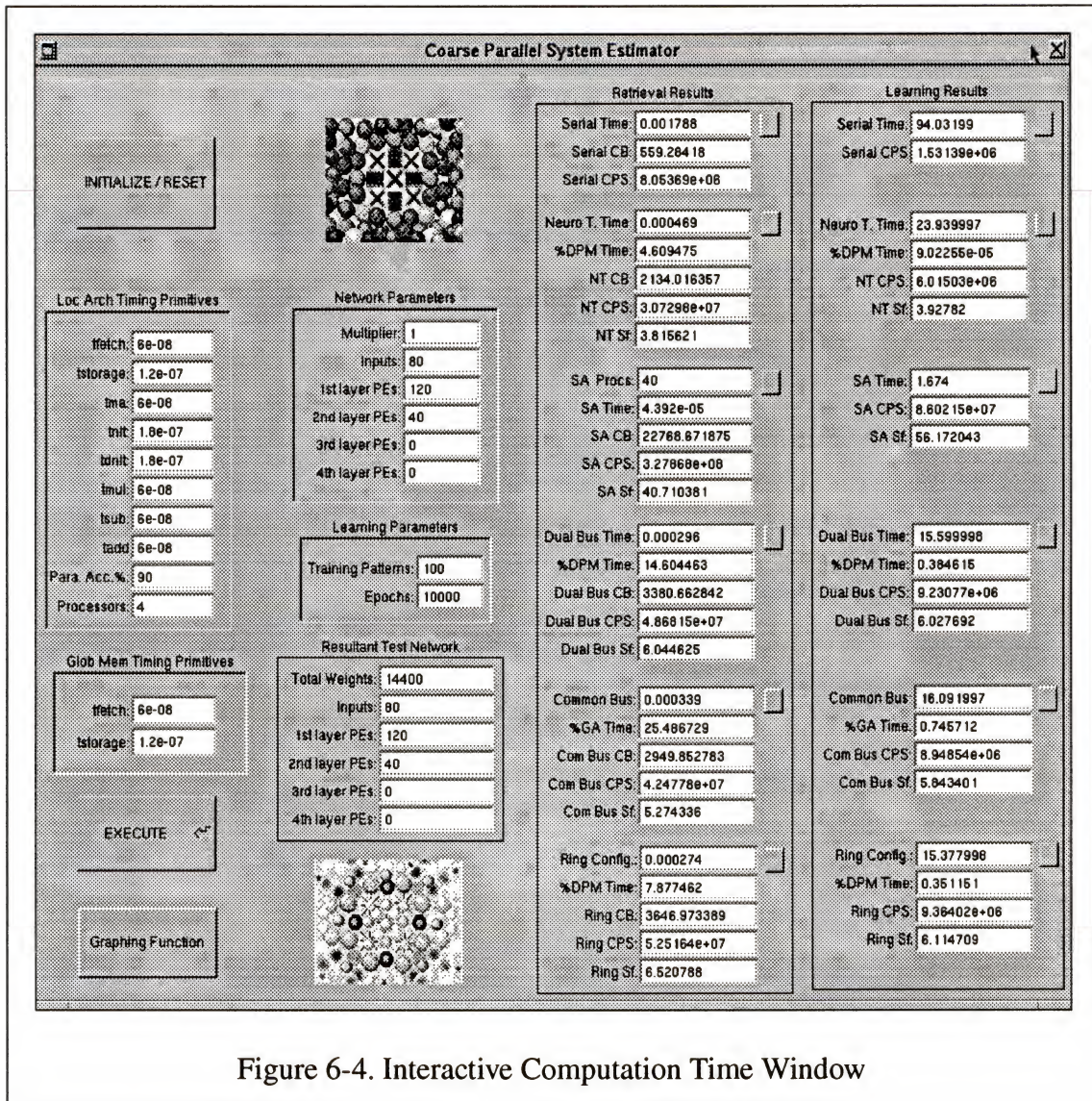
Ring Configuration Backpropagation Time (RCBT)

$$RCBT = \frac{1}{P} (SCBP) + \frac{(P-1)}{P} \sum_{k=1}^{K_L} N_k (t_f + t_s) + \frac{(P-1)}{P} \sum_{k=1}^{K_L} N_k (t_f + t_s) \quad (6-6)$$

The last term however has slightly deviated from the progression found from equation (6-4) to (6-5). The $(P-1)$ term is no longer only multiplying t_f but is now multiplying both t_f and t_s . This has changed due to the distributed memory storage of the ring. Previously, all processors were required to store their error terms in global memory. This was equivalent to storing N_k total values per k layer. Under the ring configuration the N_k terms are divided across all P processors such that N_k/P errors are stored in a single DPM. In order for all processors to see these terms (error terms generated by another processor) they must be spun through the ring $P-1$ times. See Figure 3-11 for an example of a single write cycle in a ring architecture. In conclusion, the total storage or fetch time is $(P-1)(N_k/P)$ times the unit primitive for a ring configuration. If P is again made equal to 4, equation (6-6) will produce the same results as Neuro Turbos backpropagation computation time formula (5-16).

6.5 Creating an Interactive Environment for the Computational Time Formulas

At this point, computational time formulas have been derived for Neuro Turbo, systolic arrays, a single bus/global memory architecture, a dual bus architecture and a general ring dual port memory per processor configuration. The time formulas were estimates of computation times for each architecture operating under retrieval or backpropagation. While these formulas appear lengthy, they do contain a great deal of information providing that they can be quickly computed in an interactive manner. For example, we would like to observe (across all architectures) how the total computation times change with respect to alterations in system primitives, increasing network size and/or adding system processors. The NeXT Machine's Interface Builder turned out to be an excellent platform to do just this. The time formulas were coded in 'C' and then an object C front-end was developed to create an interactive environment. See Figure 6-4. Inputs are shown in the blocks labeled Local Architecture Timing Primitives, Global Memory Timing Primitives,



Network Parameters and Learning Parameters. Outputs are shown in blocks labeled Resultant Network, Retrieval Results and Learning Results.

Beginning with the Local Architecture Timing Primitive block there are ten input windows. The top eight of the ten are timing primitives associated with the computation time formulas. These values are all in seconds and are initially set by the INITIALIZE/RESET button on the top left of Figure 6-4. The actual numbers (e.g., 6e-8, 1.2e-7, 6e-8,

etc...) were obtained from the Texas Instruments TMS320C31 technical documentation. The reasons for selecting the TMS320C31 as the base system processor will be described later.

The two remaining input windows in the Local Architecture Timing Primitive block are the Parallel Access percent and P, the system processor count. The system processor count is simply the number of base processors comprising the system. The parallel access percent however is an attempt to further tailor the computational time formulas to a processor's internal structure. For example, in equation (5-11), backpropagation through the weights and (5-15) retrieval, there are N^2 terms which multiply the sum $t_f + t_{ma}$. The internal structure of the TMS320C31 is such that the fetching of the weights (externally), fetching of the activations (internal RAM) and multiply accumulate operation can all be effectively performed in parallel. The processor's ALU has separate internal data paths for internal and external memory. The parallel fetch operations are pipelined with the multiply accumulate operation such that a single dot product can be performed in one instruction cycle. Therefore if the parallel access is set to 100%, the $(t_f + t_{ma})$ sum is reduced to t_f . The default value (initialized at start) is set at 90% which equates to $(t_f + 0.1t_{ma})$. The author selected this as the default, rather than 100%, to allow for lost time when initially filling the pipeline.

The Global Memory Timing Primitive input block has two primitives associated with global fetches and storage of data. These two primitives represent the access times for the global memory or dual port memory in the common bus, dual bus and ring configuration architectures.

The network size (PEs per layer and the number of layers) is set in the Network Parameter Block. The user can describe the network layers and then use a multiplying factor to quickly scale the network up. For example, one may want to start with a small

network and then observe the effect of scaling it up by 10, 100 and 1000. The user merely enters these values one at a time into the 'multiplier' window and observes the output for each case.

The last of the input blocks is the Learning Parameters. This has two windows which correspond to the number of epochs and training patterns (batch size). The training pattern input is the number of input/desired vectors presented to the network. This is also termed a batch. An epoch is defined as one presentation of all the vector pairs in the batch. The total number of iterations (weight updates) is therefore equivalent to the number of training patterns times the number of epochs.

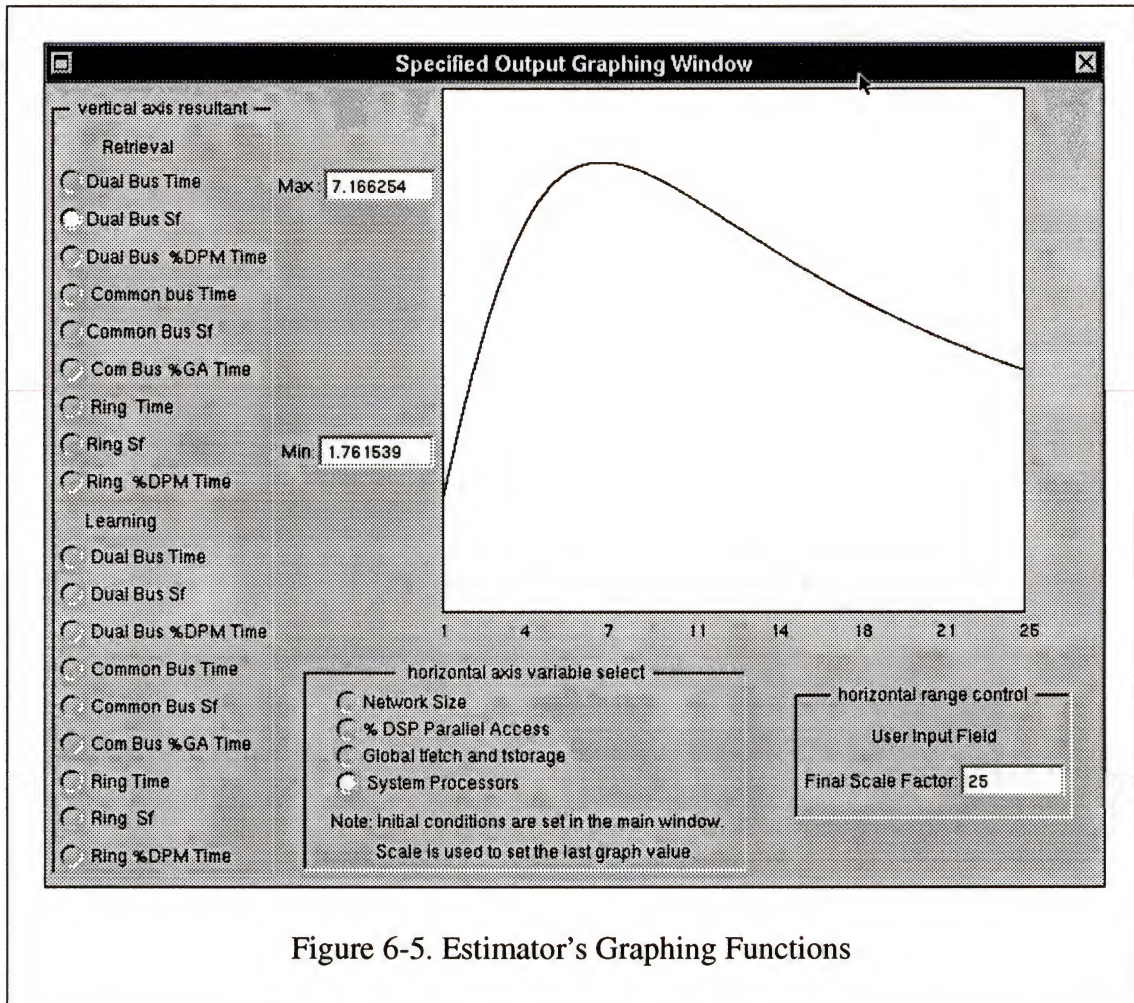
The Resultant Test Network output block was created to show the total number of weights in the network and the scaled versions of the PEs per layer. When the multiplier is used in the Network Parameter input block, the user can instantly view the PE per layer size increase or decrease resulting from the multiplier value.

The remaining two output blocks generate computation time values for both retrieval and learning on six different architectures. The topmost architecture is the serial machine which corresponds to equations (5-9) through (5-15). This single processor has only local memory and no internal parallel data paths to its ALU. The overall retrieval computation time and learning computation time is shown on top of the correspond retrieval/learning column. The 'Serial CB' stands for the serial computational bandwidth. *Computation bandwidth (CB) is $1/\text{retrieval computation time}$.* This is the estimated number of output vectors that can be generated per second during retrieval. 'Serial CPS' is equivalent to the serial connections per second. *Retrieval connections per second is defined as the number of weights divided by the retrieval computation time.* Under learning the formula slightly changes. It becomes the total number of weights divided first by the number of training patterns times the epochs and then by the serial computation time.

The learning connection per second value is therefore defined as the total number of weights divided by the time to perform one network weight update (iteration).

Computation times, bandwidths and connections per second numbers are also computed for Neuro Turbo (NT), the systolic array (SA), dual bus architecture, common bus architecture and ring configuration. The following new outputs are generated for these systems as compared to the serial machine. First, each system can now be compared to the serial machine and thus a speed-up factor (Sf) is computed. *This is the serial machine's computation time divided by the parallel system's computation time.* Second, because four of the five parallel systems contain some type of global communication, the amount (as a percentage) of global communication is compared against the overall computation time. For example in Figure 6-4, dual bus retrieval shows a '% DPM Time' equal to 14.6 percent. This says that approximately 14.6 percent of the total retrieval computation time is spent in dual port memory access. Similarly the common bus has a global access (GA) percent which corresponds to the time spent fetching/storing values from/to global memory. Third and finally, the systolic array also has a window showing the number of systolic array processors comprising the ring. Referring back to section 4.6, the total number of array processors one could apply to a particular network problem is equal to the smallest layer in the network. The estimator always assumes the user has access to this size systolic array. For example, if the smallest layer in the network is 10,000, the ring size will be set to 10,000 array processors. Thus the systolic array numbers are based on the optimum array size dictated by test network size (not by economic cost).

A graphing function was also added to the estimating software to observe how a specified output changes to increasing network size, parallel access percent, global timing primitives or the system processor count. See Figure 6-5. A user can select one of the 18 outputs listed in the column on the left of the graph. These correspond to the three



different DSP architectures previously described in section 6.3. Specifically, one can observe how the computation time, speed-up factor or percent of global communication time changes with one of the inputs found in the 'horizontal axis variable select' block. A final scale factor input area was also developed to do the following. The user types in the final scale factor (shown as 25 in Figure 6-5) and the selected input variable is scaled from 1 to the scale input by increments of the final scale input divided by 300. The initial value of the selected input variable is obtained from the value it was set in the main window. For example in Figure 6-5, the selected input variable designated for scaling is the system

processor count (P). This was originally set to one in the main window such that the first value on the graph's horizontal axis would start at one. The final scale value was defined as 25 and so 300 points between and inclusive of the endpoints (1, 25) were generated. The vertical axis in Figure 6-5 therefore corresponds to the dual bus' speed-up factor (Sf) with respect to changing system processor count. The result shown by this graph will be discussed in the next section.

In summary, software has been developed to observe how sensitive the three basic DSP architectures are to changes in the system primitives. Sensitivity to increases in network size can also be instantly viewed. The user simply alters the appropriate input parameter and seconds later the change can be observed. A graph function further aids the user to see how changing the system processor count, global primitives, percent parallel access or network size affects an architecture's computation time.

6.6 Results using the Coarse Parallel System Estimator Software

In the previous section, it was mentioned that the primitives were obtained from the Texas Instruments TMS320C31 technical notes. The 'C31' was selected as the basic system element for several reasons. The first is its internal structure which contains parallel data paths between the ALU, external memory and internal memory. The second reason is the processors pipelined data storage, fetch and multiply accumulate functions. This enables the processors to fetch two datum, perform a multiply accumulate operation in 60 nsec (based on a 25MHz C31). A detailed example of applying this feature to neural net processing was given in the last section. The last primary consideration for selecting the C31 was its low price when purchased in volume quantities. The processor is expected to cost in the range of \$35-40 when purchased in quantities of 100 or more.

Upon deciding what base processor will form the core of our parallel system, supporting memory size and structure was next considered. Local memory was chosen to be

25nsec SRAM to support the CPU transfer rate. The size of this RAM will be discussed in section 6.7. Global memory was next considered and the question became whether to use all SRAM (single bus architecture) or improve system performance with more costly dual port memory (the dual bus architecture and/or ring configuration). With this question of selecting the architecture, came other related questions. For example, which architecture best supports retrieval? Learning? How does network size affect these results? etc... To answer these questions, the estimator software was employed. In order to keep this section brief, the results will be presented in a question/answer format. Supporting results from the estimation runs will also be shown graphically when deemed necessary.

6.6.1 Comparing the Three Architectures Operating in Retrieval

For retrieval processing is there a difference between the three types of DSP architectures? To answer this a simple network was first input to the main window. This network consisted of 8 inputs, 12 hidden layer PEs and 4 output PEs. The system processor count P will be fixed at eight which is considered a mid-size system. Normally, four is a small system and ten or more processors a large one. The global access times were fixed with $t_{\text{fetch}} = 6\text{e-}8$ and $t_{\text{storage}} = 1.2\text{e-}7$ seconds. These were obtained directly from the technical specs of a currently available popular dual port memory chip. Having set the elementary primitives, the graphing window was then used to show how the three speed-up factor changes with network size for the three different architectures. See Figures 6-6 through 6-8. Several important observation can be made. The first is that the rate at which the ring configuration reaches its maximum value is much faster than either the common bus or dual bus architectures. This is due to its reduced global communication time when compared to the other two systems.

However is this a significant difference to warrant placing a dual port memory between each processor? To answer this we must first look at all three curves from the

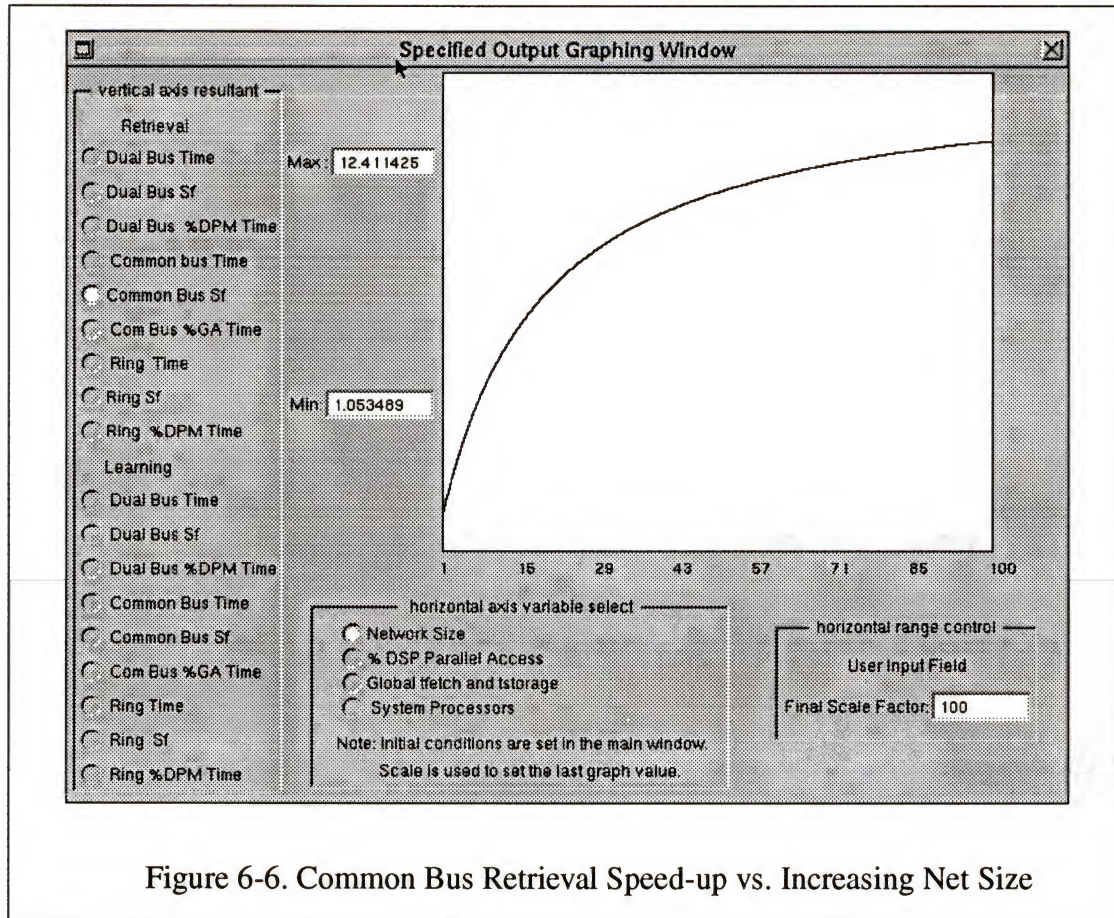


Figure 6-6. Common Bus Retrieval Speed-up vs. Increasing Net Size

point when the network has been scaled-up 50 times or more. The dual bus speed-up (roughly 12) appears to be very near the ring (approximately 14) and the common bus is closely behind at around 10. From this point on also, the differences appears to be minimal. Thus little will be gained by employing a ring architecture for nets larger than 50 times our original 8 input, 12 hidden PEs and 4 output PEs (360,000 total weights). For networks less than this size, it turns out that retrieval will execute on all three of the machines faster than most applications should require. This was determined by looking at the computational bandwidth of the three architectures for the 50x network. The main window was used to obtain the following results. The computational bandwidth for the

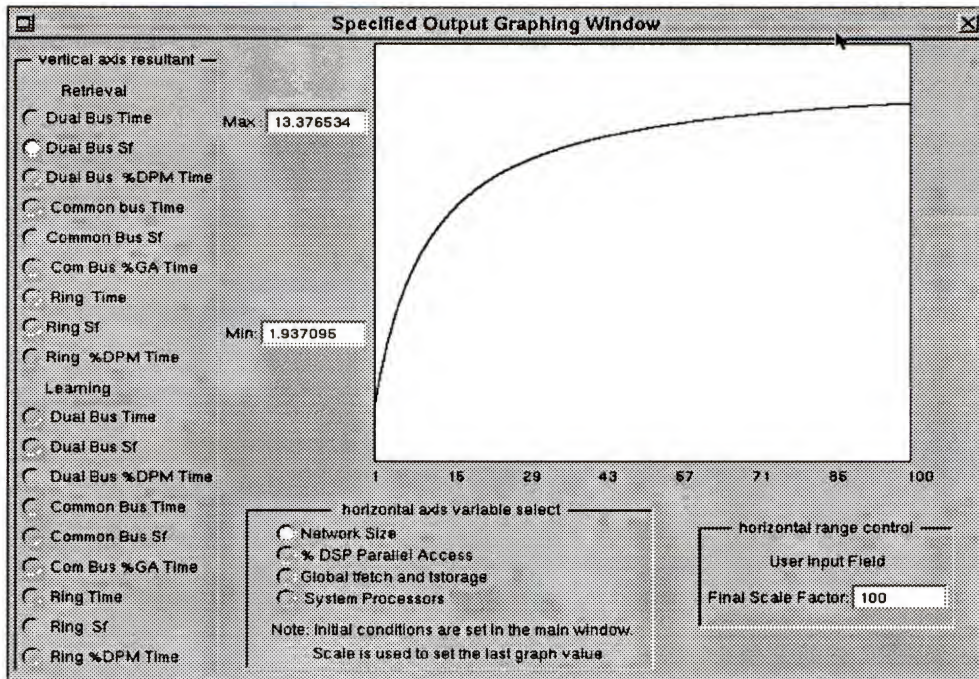


Figure 6-7. Dual Bus Retrieval Speed-up vs. Increasing Net Size

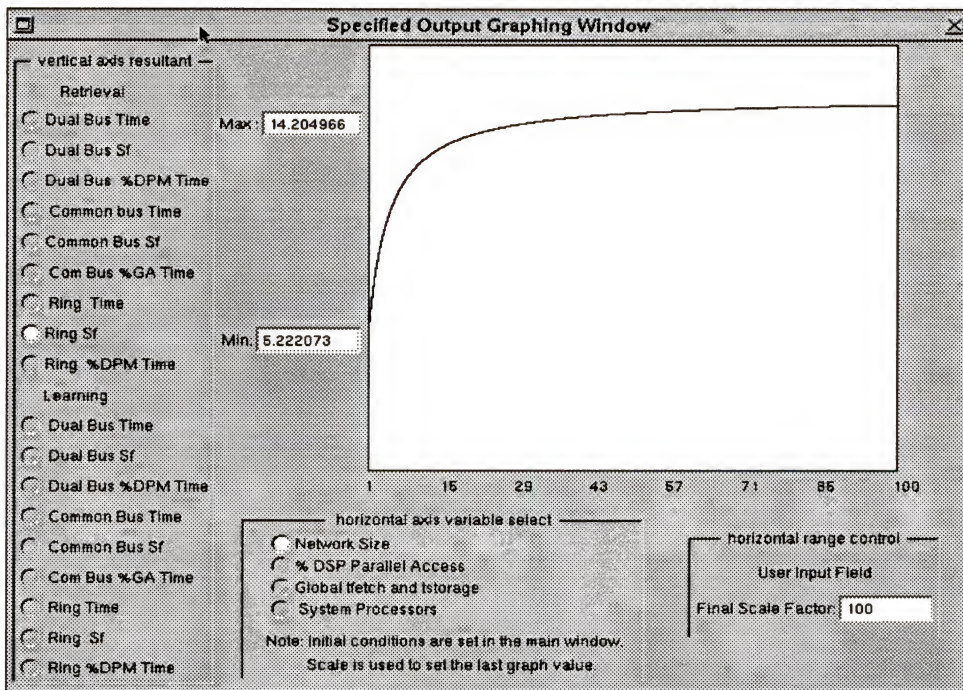


Figure 6-8. DSP Ring Retrieval Speed-up vs. Increasing Net Size

common bus is 249 output vectors per second, 284 vectors per second for the dual bus and 319 vectors per second for the ring. Any of these three systems (and associated computational bandwidths) should be fast enough classifiers for most neural applications. In summary, for retrieval, there is little advantage in using a dual bus or ring architecture versus a common bus.

6.6.2 Comparing the Three Architectures Operating under Learning

Under learning is there a difference between the three types of DSP architectures? Since there is little significant difference between the three architectures operating in retrieval, it stands to reason that learning on each system should have even closer performance results. This is due to the notion that the global communication during learning is expected to be even smaller when compared to the overall learning time (e.g., backpropagation operation, local weight update process). See Figures 6-9 through 6-11 for the learning speed-up factors. In all three cases, the architectures reach their maximum performance speed-up value more rapidly than in the retrieval graphs. This is again due to the large portion of local processing time in learning when compared to global communication. Retrieval, having a much smaller local processing time, is more sensitive to global communications between system processors. Because of this, the final maximum speed-ups in the three architectures are all very closely grouped between 12 and 12.6. Thus for moderately large networks (100,000 weights plus) there is little difference between the three systems.

For smaller networks, closer observation is required. Returning to the main window, the network scale factor is first set to 25 and then the speed-up factors for the three architectures are recorded. (new net = 25x8 inputs, 25x12 hidden layer PEs, and 25x4 output PEs) For reference purposes, this network contains 90,000 weights. The speed-up factors for the three systems (common bus, dual bus and ring) are 11.2, 11.8 and 12.3. This is

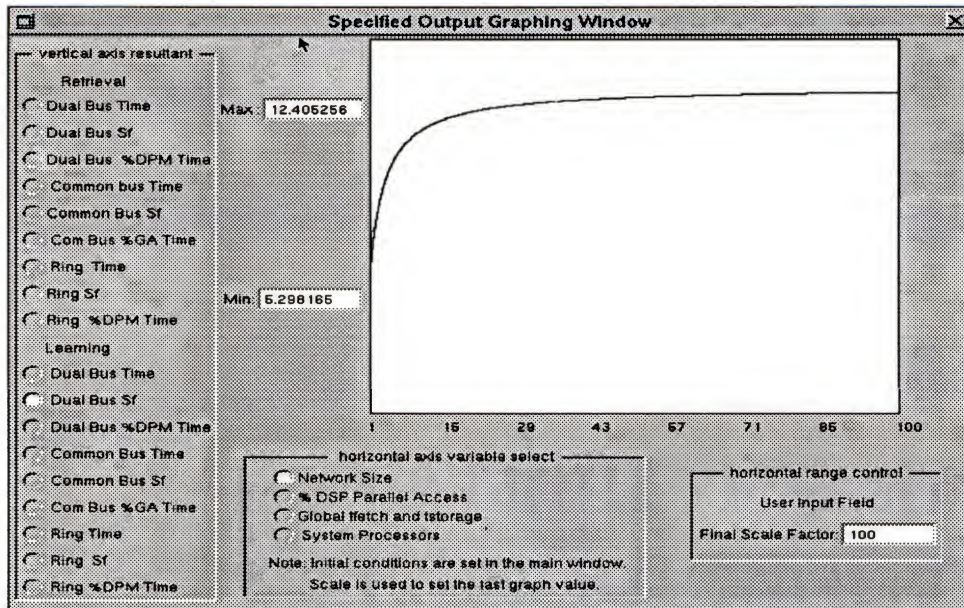


Figure 6-9. Dual Bus Learning Speed-up vs. Increasing Net Size

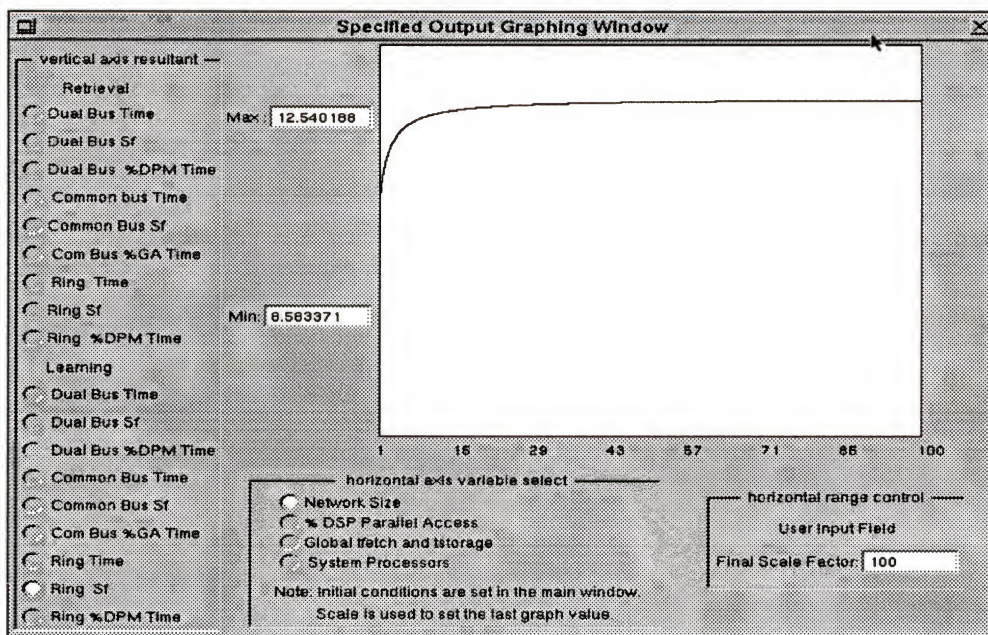


Figure 6-10. DSP Ring Learning Speed-up vs. Increasing Net Size

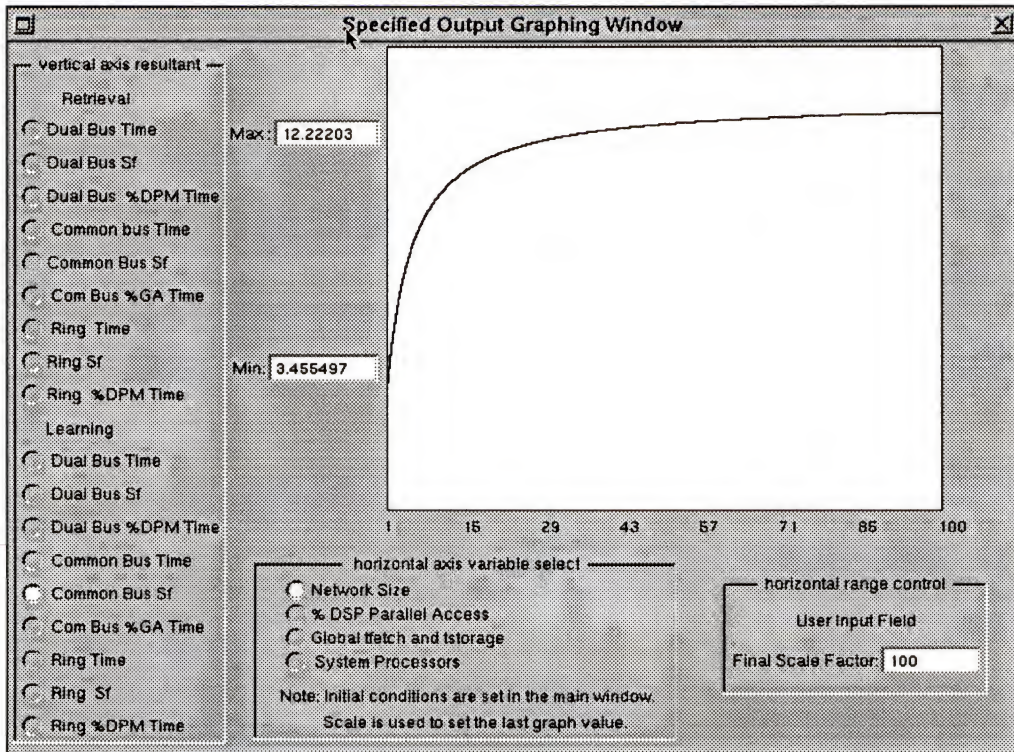


Figure 6-11. Common Bus Learning Speed-up vs. Increasing Net Size

not a significant enough difference to warrant selecting the ring or dual bus over the common bus. Hence the scale factor is further reduced to 15 or 32,400 weights. The new speed-up factors are now 10.4, 11.4 and 12.2 for the common bus, dual bus and ring. If we also set the batch to be 1000 training vectors and set the epochs to 10,000 (typical learning parameters), one can observe the overall learning time associated for each of the architectures. This turns out to be 5.6 hours, 5.14 hours and 4.8 hours for the three different architectures. Again this does not appear to be a significant difference for selecting a particular system versus another.

Reducing the scale further to 10 times or 14,400 total network weights, the speed-up factors for the three systems begin to deviate slightly. The values are 9.6, 10.8, and 11.9

for the common bus, dual bus and ring systems. If we return to the 1,000 training vector batch and 10,000 epochs case, the learning times are 2.7, 2.4, and 2.2 hours for the three corresponding architectures. This still is too small a difference between the three to warrant using dual port memory.

It turns out that the original network (8 inputs, 12 PEs, 4 PEs) must be scaled in the neighborhood of 2 times (576 weights) before a significant dip in performance can be observed between the ring and common bus architectures. Using the estimator's main window, the three speed-up factors read off as 5.15, 7.2, and 10.0. The ring is therefore almost twice as fast as the common bus architecture for this size network. The learning times associated with the 1,000 training pattern/10,000 epoch example are now 13.1, 9.4, and 6.7 *minutes* for the common bus, dual bus and ring. While this is a significant difference in computational efficiency between the architectures, the question now becomes whether it's worth saving a few minutes to go to a more complex bus design.

In summary, the performance of all three architectures is nearly identical during retrieval or learning for large networks (100,000 weights and above). This is in essence due to the local processing being so large as compared to the global communication between system processors. *The performance between the common bus and ring does however begin to differ for smaller networks (i.e several hundred weights or less).* This is not a problem for a common bus system operating under retrieval mode only. The network is small enough that several thousand classifications can be performed in a second.

In learning, however, the common bus was shown to be twice as slow as the ring for a network containing 576 weights. Recalling the example where the scale was 2 and the learning consisted of 1,000 patterns and 10,000 epochs, the common bus computation time was 13.1 minutes. The ring was found to be roughly half of this; however, because both these times are small the difference was not considered to be significant. The notion

of waiting 13 minutes rather than 6.5 minutes for a result would not upset too many neural researchers. However this could become a problem if the previous learning example was changed to 10,000 patterns and 100,000 epochs (an increase of 100 times). The common bus learning time estimate increases to 1,310 minutes or 21.8 hours. With the ring estimations at half this value, the user might consider developing a ring to yield results over night. However the 10,000 pattern/100,000 epoch learning example is an unusual case and is only presented as a potential pitfall to selecting the common base architecture. The 1,000 patterns/10,000 epochs is far more typical. Thus for the vast majority of multi-layer perceptron neural networks, performance differences between the three architectures should be minimal.

6.6.3 Optimizing System Processor Count for the Common Bus, Dual Bus and Ring

Previously, the three simple DSP configurations were compared against one another for retrieval and learning. The results showed that only for a very specific case (a small network, large training pattern file and large number of epochs) did the performance of the common bus not track the ring. These results were based on a simple network containing 576 weights, partitioned across 8 system DSPs. Upon further investigation, however, it was also observed that smaller nets will run *more or less efficiently on the common and dual bus architectures depending on system processor count*. See Figures 6-12 and 6-13. In these two figures, the 576 weight network was kept constant while the system processor count (P) was varied from 1 to 40. The common bus appears to show an optimum speed-up value at 7 and the dual bus around 8. The dual bus, having a more efficient global communication structure, can throw one more processor at the problem without losing overall computational efficiency. In the case of the ring, even more processors can be added to reduce the learning time. See Fig. 6-14. The ring does not show a peak speed-up

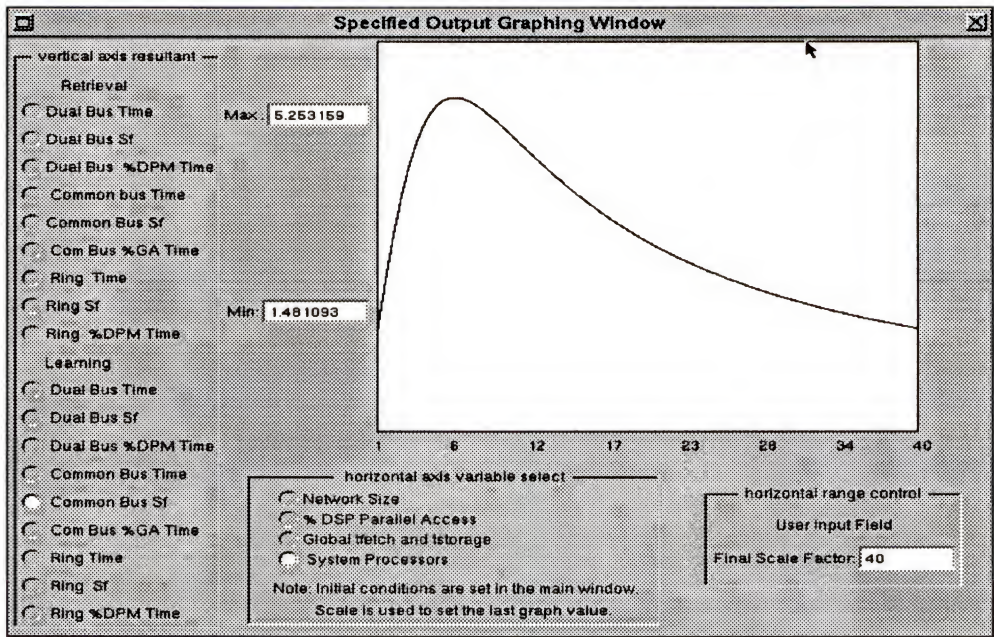


Figure 6-12. Common Bus Learning Speed-up vs. System Processor Count

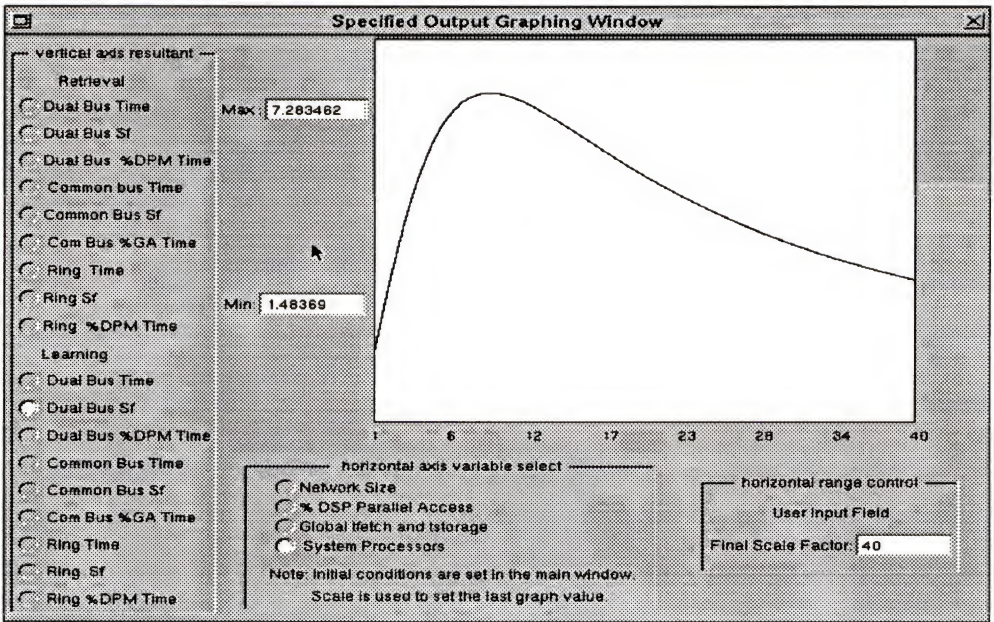


Figure 6-13. Dual Bus Learning Speed-up vs. System Processor Count

factor versus the increasing system processor count. However after a couple hundred processors have been added to the system it tend to flatten out. See Figure 6-15.

Thus the learning times presented in sections 6.6.1 and 6.6.2 were optimal or very near optimal for both the common and dual bus systems. However the ring could have been doubled to further reduce the overall learning time. Hence if it is logistically possible to build a system with 40 DSPs in a ring, the speed-up could be as large as 29.3 which dwarfs the common and dual bus optimum speed-up factors. The feasibility of developing a system will be discussed further in section 6.7.

6.7 The CNEL Machine

Upon testing several different size networks with the estimator, it became apparent that the learning gains associated with the DSP ring were limited when compared to the other two architectures. For large networks (10,000 weights) and typical learning situations (less than 5000 training vector pairs and 10,000 epochs or less), the learning speed-up factors for the three different architectures are very similar. However architectural design simplicity is not equivalent for the three types of systems. The ring has many more logistic problems. For example, when considering what type of system to construct, the ability to expand was a major issue. We wanted to design a system that would start with a basic model (2 DSPs) and expand up to a full loaded machine with possibly 12 DSPs. This would therefore allow a wide range of users to tailor a system to their current needs. In addition, low end users could add boards as their research problems grew or as additional funds became available. A ring is less easily expandable as compared to the common bus or dual bus system. This is due to the closed loop structure required by the ring architecture. The first problem that arises is what type of backplane would support this type of design. The architecture suggests a circular design which is atypical and therefore costly. In addition, if the ring system is designed to support a maximum number of base

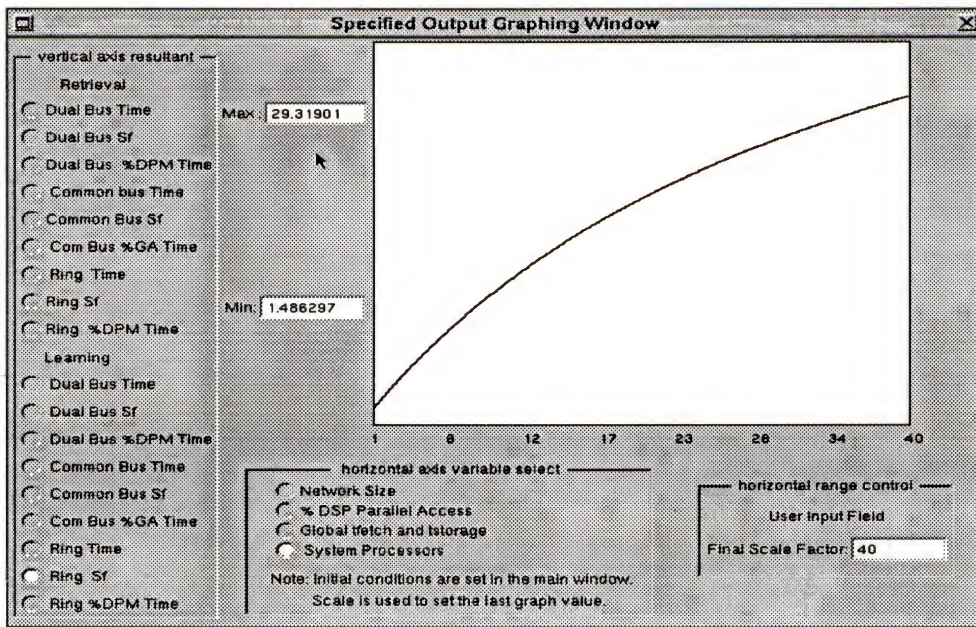


Figure 6-14. Ring Learning Speed-up vs. System Processor Count

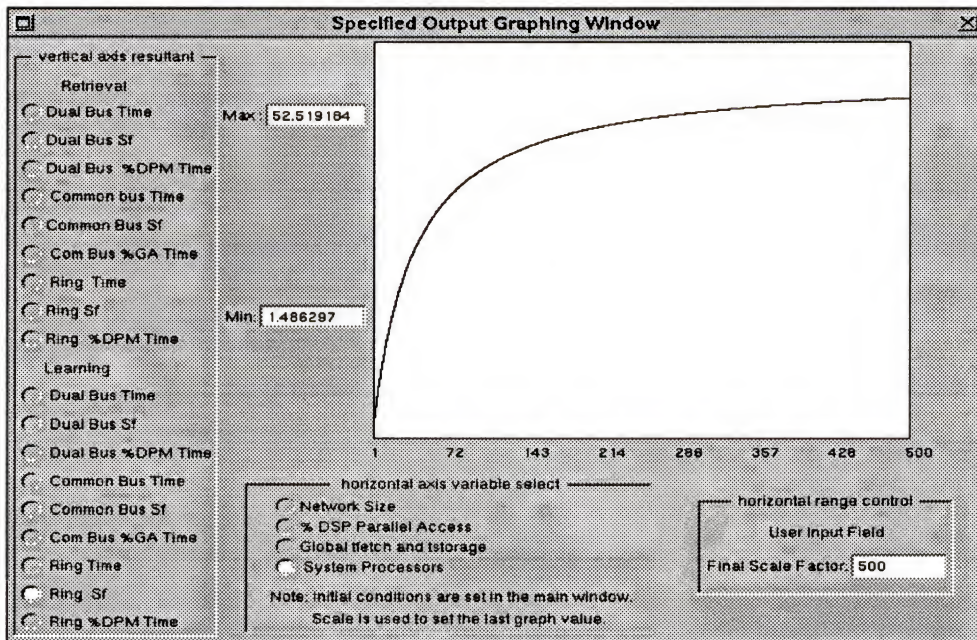


Figure 6-15. Ring Learning Speed-up vs. System Processor Count (Pmax=500)

processors (e.g., $P=12$) there is the question of how the system is loaded for smaller configurations (e.g., $P=4$). Most likely an additional card would have to be developed (fabricated) to act as a dummy connect card in order to pass signals through the empty processor socket to an existing base processor. This again adds to the overall system cost. e.g., 8 dummy cards for a system containing 4 base processor cards.

Another problem with the ring is the cost of having P dual port memories (DPMs). Dual port memory currently is several times as expensive as generic SRAM. Because of this DPM expense and the complex backplane design requirement, the ring was not considered for construction. The common bus and dual bus architectures have much more simpler backplane requirements. Both systems have open ended buses. This is a much more traditional busing structure common to both work stations and PCs. Processors can be easily added to the buses in either case. An additional processor is simply added to the next open slot in the backplane. Where this becomes a problem is in connecting several processors (greater than 6) to a single bus. The capacitive loading on the bus becomes so great that signals can't be propagated to there destinations in the required bus cycle.

Because of this reason, the dual bus was chosen over the common bus. We wanted a system expandable up to 12 total processors rather than the limit of 6 for the common bus. See Figure 6-16. This will be referred to as the Computational NeuroEngineering Lab Machine or CNEL Machine for short. Local memory consists of fast access (25 nsec) SRAM and the base DSP is the TMS320C31. The block residing between the global bus and DSP is the bus arbitration hardware (PALs and TTL glue) required for bus sharing. It is important to point out that the arbitration block has replaced the dual port memory in the ring configuration. In the ring, the dual port memories simplify the memory access such that no arbitration is required. However simplicity in hardware design equates to a higher economic cost (the expense of each additional DPM per DSP). Instead we chose to expend

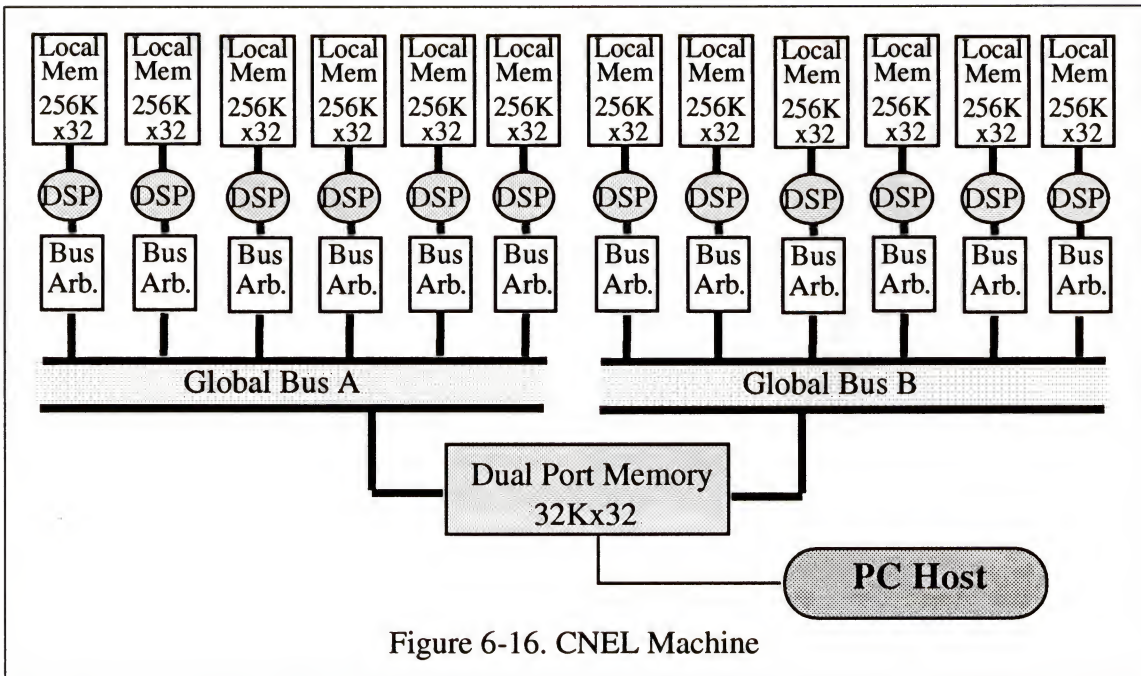


Figure 6-16. CNEL Machine

time simulating arbitration hardware (e.g., PAL state machine design) rather than money for components.

The memory sizes arise both out of the base processor constraint (C31 is a 32 bit engine) and the neural algorithm. Recalling that only the partial nets or errors (both N quantities) are required to be transmitted to/from the DPM, the DPM can be kept relatively small. A base quantity will probably be $8K \times 32$ bits expandable to $32K \times 32$ in 8K sections. Local memory must hold N^2 weights and therefore is much larger when compared to the DPM block. A basic block will contain $64K \times 32$ and be expandable up to $256K$ words. Hence memory is also variable as was processor count.

Having chosen this type of system, one last check was made for peace of mind purposes. The dual bus learning speed-up factor was computed in the case where the dual port memory (global communication) access times were increased 10 times from those shown in Figure 6-4. The number of processors were again fixed at $P=8$ and the original test

network was scaled 10 times (14,400 weights). For this factor of 10 degradation in global communication, the speed-up factor was found to drop by 50% for the moderate size network. At first this was considered to be unwelcome news. However, it was found that after P was lowered to 4 system processors, the speed-up factor was not nearly as sensitive as the 8 processor case. The resulting conclusion being that depending on the final global (DPM) memory access times, certain size networks will run more efficiently a smaller group of P processors. Systems with 8-12 processors will only be required for networks with 100,000 weights or more. In any event, if the user knows the system primitives (or is given them after purchasing a system), the optimum processor count can be obtained from the Coarse Parallel System Estimator's graphing function.

6.8 Chapter Summary

In the beginning of this chapter four main goals were laid out as a grounds for selecting a particular parallel system. They were that the system be computationally flexible, inexpensive, fast and readily available. With respect to computational flexibility, the base processor used in the system was selected to be a DSP. This type of processor has a flexible instruction set that should be able to accommodate a wide spectrum of pre-processing functions (FFTs, low pass filters, correlation, etc...). Specifically the TMS320C31 was selected as the DSP of choice because of its low unit cost and internal parallel data paths.

Upon selecting the processor, three different simple DSP configurations were tested (simulated in software). These were the common bus, dual bus and ring architectures. The performance of these three systems, under average learning conditions, appeared to be very similar for moderate to large networks. Because of this uniformity in computational performance, the three systems were next critiqued for simplification in design and construction. The ring turned out to far more complex to fabricate than either

the common or dual bus configurations. The dual bus was then selected over the common bus for electrical purposes. The common bus would not support the maximum desired number of base processors in the system. Because the dual bus system contains only one small dual port memory block it is expected to be a relatively inexpensive system to realize when compared to the ring.

In terms of improvement in speed, the introduction in section 6.1 mentioned a desired benchmark speed-up of 50 over the NeXT computer. This will now be briefly discussed. Several timing measures were taken on the NeXT to determine an average primitive time for a typical single dot-product operation. This operation consisted of fetching the inputs, weights, multiplying the two values, and adding the result to the sum in the 68040's ALU. In the best case (a 33 MHz turbo NeXT Machine executing several thousand dot-products in a loop) the time to perform this operation averaged out to be 2.44 micro-seconds under retrieval. For backpropagation, a very similar operation, the number equated to 2.2 micro-seconds. If we return to the C31, the internal structure of the processor is such that we expect the weight fetch, input fetch and multiply-accumulate to be executed in 60 nano-seconds (e.g., the external bus speed, 16.7MHz). This is based on the assumption that the input values (activations and errors) can all be stored internally in the processor and weights are stored externally in fast SRAM. Dividing the turbo NeXT's best figure of 2.2 micro-seconds by 60 nano-seconds, a speed-up figure of 36.7 is obtained for a system based on a single 'C31'. Assuming that a four processor system can be constructed, the speed-up jumps to 146.8. This however is a best case estimate. In reality there will probably be a certain amount of overhead code required by the DSP function (e.g., variable indexing) and global communications will also further decrease efficiency. Thus for a more conservative estimate, a speed-up factor of 100 will be estimated for a four processor system which greatly beats our initially desired 50 speed-up factor figure.

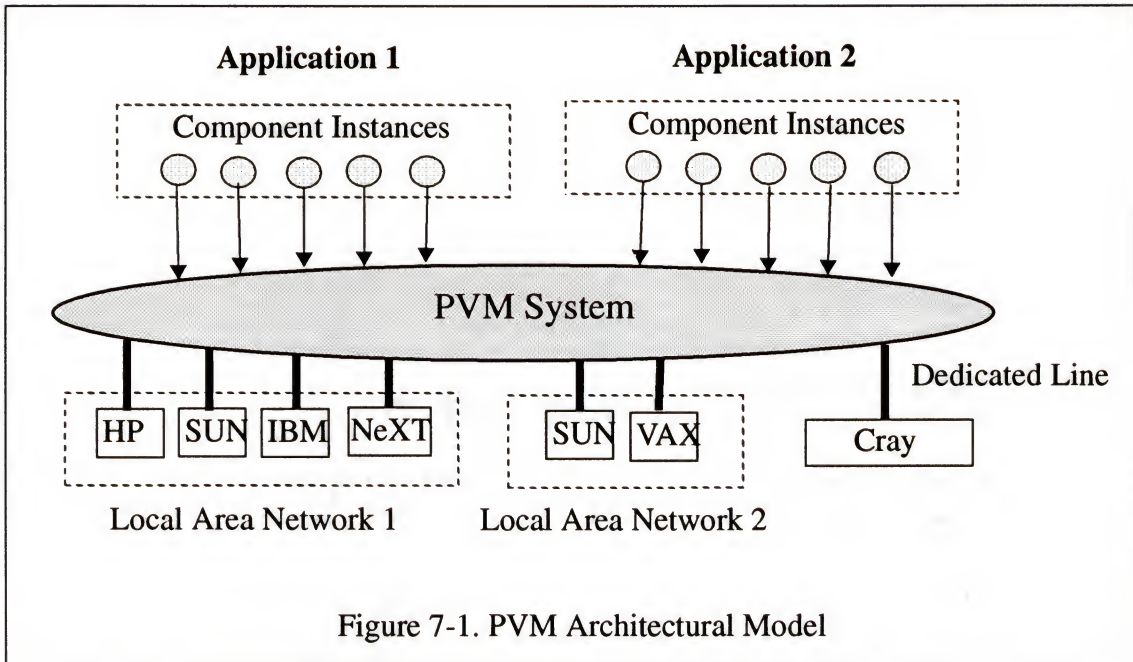
CHAPTER 7 WORKSTATION CLUSTERING

7.1 Introduction

During the same time frame that multiple DSP system research was being performed, a software workstation clustering package became available. This package is known as the Parallel Virtual Machine or PVM for short. It is a shareware program that enables a collection of heterogeneous computer systems to be used as a coherent and flexible concurrent computation resource [43][44][45]. The individual machines may be shared- or local-memory multiprocessors, vector super computers, workstations or other UNIX based machines. PVM provides drivers for NeXT Machines, SUNs, IBM R6000s, HP computers and most other popular UNIX/UNIX variant computers.

The function of PVM is to give a single user or host the ability to use several other machines (called nodes) concurrently; thus forming a single parallel system. A major feature is that PVM transparently handles message routing and data conversion between incompatible architectures. For example, the user can build a cluster of SUNs and R6000s and not have to worry about incompatible communication protocols between the two different types of systems. The only requirement is that all machines be connected via some kind of network (e.g., Internet, local Ethernet, fiber optics, etc...). PVM is therefore considered a *coarse parallel distributed system with loosely coupled connectivity*.

PVM is comprised of two main parts; a daemon process and a user library. The daemon process is used to build the parallel system in the following manner. For each type of machine, SUN, HP, NeXT or other UNIX/UNIX variant machine, the daemon is first compiled (via a make file) to form the executable. Upon compiling, the daemon



executable is then copied into the temporary directory on all machines designated to form the cluster. Once there, it is run in the background of each cluster member. PVM then lays dormant until commands activate it from a particular user program. This brings us to the second portion of PVM previously mentioned, the user library. This is a super set of 'C' commands which are compiled and linked in with a normal C program. These commands enable one to initiate processes on other machines, communicate between processes and process synchronization.

An application is therefore comprised of components (subtasks) that can be executed on any machine in the cluster. When the component is executed on a machine, it is referred to as an instance of that component. Hence a particular routine may be executed multiple times (multiple instances) on one or more machines. See Figure 7-1. This turns out to be a very powerful and easy to use construct. The user first writes a host program

and the host then initiates instances of a particular component (node programs) on a slave or node machine.

In addition, PVM was developed from a government grant to the Oak Ridge National Laboratory and thus is considered shareware (free to anyone in the USA). To obtain PVM and/or supporting documentation, simply type a note with 'send index from pvm' and email it to 'netlib@ornl.gov'.

7.2 Clustering NeXT Machines

Providing that a user has access to one or more remote UNIX-like workstations, one can experiment in parallel processing with no additional hardware costs. In fact, because PVM was obtained for free, the only expense incurred is the user's own software development time. In the author's case, this turned out to be relatively minor due to PVM's ANSI C like instruction syntax. However the down side of using PVM is that the speed-up factor of this parallel system is relatively small. There are several limitations when using this system. The first is the number of systems one can use. Usually only a few workstations can be employed on a problem before inter-processor communication (Ethernet bottleneck) becomes too significant. Another problem is the type of workstation used. It is rare to find a homogeneous cluster in either academia or industry. Typically, one must choose from a group of heterogeneous machines where certain machines may be much slower than others. Because the speed-up factor is based on one machine, it must be based on the fastest machine one has access to in the cluster. Because of these two limitations, the author felt the highest speed-up factor one could obtain using PVM in his lab would be far less than the desired 50 number discussed in the last chapter. However it was decided that PVM would provide an excellent tool for testing network partitioning and for testing the prediction equations generated back in Chapter 5.

The Computational NeuroEngineering Lab (CNEL) contains 6 NeXT Machines networked via a local Ethernet bridge. In addition, the author had access to 2 remote Turbo NeXT Machines and a non-turbo all which resided in professor's offices. Of the nine machines available, however, only 6-7 maximum were usually used for clustering. This due to the following problems. One of the NeXTs (named Synapse) was configured as a dedicated file server to a 1 giga-byte disk for all the other NeXTs. Because of this, it was found to be nearly twice as slow as a turbo machine and 1.5 times slower than the average non-turbo NeXT. Similarly, another machine (Bacchanal) was configured to be a print server for the network and was also found to be significantly slower than the average non-turbo machine. The configuring of these machines for specific network related tasks required processes to be time shared in the background on these machines. This time sharing of processes therefore slightly degraded the system's performance and also made it much more difficult to use in predict performance during neural algorithm testing.

In addition, another machine (Brain) could not be used because it was constantly being removed from the network. Brain was being used for hardware development (internal multiple DSP board) and it was deemed necessary to isolate it from the network. This was done because during the early stages of hardware testing, Brain was found to 'lock-up' in the middle of a test.

In summary, although 9 machines were theoretically available, only 6 or 7 systems were used at a time in a cluster. The lack of uniformity in performance with Synapse and Bacchanal when compared to the other machines made predicting execution times (testing the neural time models) much more difficult when they were used. Thus even though all machines were NeXTs, the lab's network should be considered a heterogeneous environment. The machines were not the same when looked at from a performance point of view. In conclusion, having different performance machines in a cluster will directly affect how

an algorithm should be partitioned across the network. Depending on communication times, it may be better to not use certain machines and create cluster out of a few select *fast* systems.

7.3 Initiating PVM and Example Code

The purpose of this section is to briefly introduce how to create a computer cluster and use the PVM 'C' library. In addition, the neural net partitioning scheme will also be presented and global communication constraints will be described. The results from using this software (computation, communication times and speed-up factors) are presented in the last chapter.

7.3.1 Creating the PVM Cluster

As mentioned earlier, the pvm daemon must first be launched before cluster programs (host and node) are activated. This was done with the UNIX script file shown in Figure 7-2. A remote shell is first launched for a particular machine in the network and then a directory titled "pvm" is created off of the temporary directory. Once this directory is created, the daemon's executable is copied from the PVM source code directory. The daemons are then all then started by the host code shown in the bottom of Figure 7-2. This code basically starts a daemon on the host (out of the source code directory) and uses a file called "host_names" to tell the host daemon which machines to log in (start-up) the cluster. See Figure 7-3. The actual cluster is formed out of four lines near the end of Figure 7-3. These machines are iris, barney, andrew, and chaos. An "*" comments a line out. The host in this case is "iris" and must be the first machine in the list. The block above this list is used by the neural net application program. The neural application program uses the numbers associated with a particular machine (e.g., Iris=1.0) to partition a network according to a system's performance. For example, sac and brain have lower performance numbers and thus should receive a smaller weighted portion of the network. This was an

```

echo "iris"
rsh iris "mkdir /tmp/pvm"
rsh iris "cp ~/pvm/pvm2.4/src/NEXT/pvmd /tmp/pvm/pvmd"

echo "barney"
rsh barney "mkdir /tmp/pvm"
rsh barney "cp ~/pvm/pvm2.4/src/NEXT/pvmd /tmp/pvm/pvmd"

echo "andrew"
rsh andrew "mkdir /tmp/pvm"
rsh andrew "cp ~/pvm/pvm2.4/src/NEXT/pvmd /tmp/pvm/pvmd"

echo "chaos"
rsh chaos "mkdir /tmp/pvm"
rsh chaos "cp ~/pvm/pvm2.4/src/NEXT/pvmd /tmp/pvm/pvmd"

echo "neural"
rsh neural "mkdir /tmp/pvm"
rsh neural "cp ~/pvm/pvm2.4/src/NEXT/pvmd /tmp/pvm/pvmd"

echo "sac"
rsh sac "mkdir /tmp/pvm"
rsh sac "cp ~/pvm/pvm2.4/src/NEXT/pvmd /tmp/pvm/pvmd"

echo "synapse"
rsh synapse "mkdir /tmp/pvm"
rsh synapse "cp ~/pvm/pvm2.4/src/NEXT/pvmd /tmp/pvm/pvmd"

echo "bacchanal"
rsh bacchanal "mkdir /tmp/pvm"
rsh bacchanal "cp ~/pvm/pvm2.4/src/NEXT/pvmd /tmp/pvm/pvmd"

~/pvm/pvm2.4/src/NEXT/pvmd -i host_names

```

Figure 7-2. Script File to Launch PVM Daemon

attempt to balance processing across all machines in the cluster. The results of this will be discussed later.

7.3.2 Partitioning a Network Across a Cluster of Computers

The partitioning scheme used for the computer cluster was essentially the same as the one described in section 4.5.1. This scheme was termed partitioning according to back-propagation alignment. The weights are split amongst the workstations according to Figure 3-4. The PE activation computations are evenly split among the cluster members such that a layer of PEs divided into even groups as found in section 4.5.1. During retrieval,

```

** "host_names"
** Host file for pvm machine cluster initiation.
** The 1st line is the host; nodes follow on other lines.
** PROCESSOR ORDER SHOULD BE THE SAME IN BOTH LISTS.

** This file sets up the following processor initiation sequence
** and processor weighting:

** 0 iris=1.0
** 1 barney=1.0
** 2 andrew=1.0
** 3 chaos=1.0
** 4 bacchanal=1.0
** 5 neural=1.0
** 6 sac=0.9
** 7 brain=0.8
** 8 synapse=1.0
** *****

iris
barney
andrew
chaos
**synapse
**neural
**sac
**brain
**bacchanal

```

Figure 7-3. List of Cluster Machines

partial sums are passed between the systems (global communication overhead) for final summation and non-linear transformation by a NeXT Machine. For backpropagation of the error through the weights, errors terms are passed via Ethernet between systems.

Although this was originally designed for a ring architecture, it was easily adapted for the workstation cluster. The cluster should be thought of as a common bus architecture where global memory is replaced with message passing (Ethernet link). Global memory access times will be replaced with Ethernet communication times. Thus, the computation time equation found in (6-4) remains nearly the same with global memory fetch and store primitives being replaced with Ethernet communication primitives. In order to give an example confirming this, the neural application software will first need to be explained.

The program was broken into two portions, a host program and a node program. The function of the host was to read in the network header (e.g., number of layers in the net, PEs per layer, input and desired file names, etc...) and then activate the appropriate nodes, thus forming the cluster. The node program would first create random weights (or read them in from a file) and then start training. All message passing was performed by sending results to the host and then having the host distribute this information to the nodes. For example, during retrieval, each node would send $N_k - N_k/P$ partial sums to the host. This equated to $(P-1)N_k$ writes to the host which matches the number found in (6-4). The host would then send all the N_k/P partial sums need by a particular node. This equated to $(P-1)N_k/P$ values to be send to each node or $(P-1)N_k$ total partial sums to be sent to the nodes. Again this matches the result found in (6-4). A similar result was found for distribution of the errors for backpropagation by the host. In summary, the global fetch plus global store time ($t_f + t_s$) in (6-4) is now replace with a primitive corresponding to sending one value between two workstations via Ethernet.

7.3.3 Host and Node Example PVM Code

In both the host and node programs, a statement is first required to enroll the program into PVM. This alerts the PVM daemon running in the background that this is a program containing PVM commands. See Figure 7-4. The host program was titled “H_net”

<u>Host Program</u>	<u>Node Program</u>
/* PVM Enrollment */ me = enroll("H_net")	/* PVM Enrollment */ me = enroll("N_net")
Figure 7-4. Enrolling a Program in PVM	

Host Program

```

/* Initiate Node Programs */
for(i=0;i<procs;i++){
    inst[i] = initiateM("N_net", &machine[i][0]);
}

```

Figure 7-5. Initiating Node Components

and the node program was named “N_net”. Upon issuing this command, the host must then initiate the node components as shown in Figure 7-5. This creates an instance out of the component or node program. In this example, a loop was used to initiate a particular ‘i’ instance on a particular machine. The variable ‘machine’ is an array of machine names which match the machines in the “host_name” file. procs is the number of processors in the cluster. After enrolling and initiating is performed, communication can now proceed between the host and nodes.

The first communication example presented consists of the host sending out network information to each of the nodes. See Figure 7-6. Each instance of a node component is addressed via the variable ‘i’ by indexing from $i = 0$ to $i < \text{procs}$ (the number of processors). To send information to a specific instance (node), an `initsend()` command is first issued to clear the PVM buffer. Upon completing this, the cluster size, number of layers, learning rate, input file size, desired signal file size and batch runs numbers are put into the buffer for the corresponding node to later remove. In addition, the partitioned PE numbers (which PEs go to which node) are placed in the buffer. Finally at the end of the transmission a variable called `msgtype` (message type) is set for use in the send statement. This tells a node which message should be removed from the buffer (e.g., the buffer may hold

Host Program

```

/***** SEND NETWORK INFO TO NODES *****/
for(i=0;i<procs;i++){
    init send() ; /* Clear Messages */
    putnint(&procs,1); /* Cluster Size*/
    putnint(&no_layers,1); /* No. of Layers*/
    putnfloat(&mu,1); /* Learning Rate*/
    putnfloat(&inp_len,1); /* Inputs (file size)*/
    putnfloat(&des_len,1); /* Desired (file size)*/
    putnint(&runs,1); /* No. of Batch Runs*/

    for(z=0;z<=no_layers;z++){
        putnint(&layer[z],1); /* PEs Per Layer */

        msgtype = 1;
        snd("N_net",i,msgtype);}
}

```

Figure 7-6. Transmission of Network Information by the Host

many messages waiting to be received from the buffer). The send statement also acts to signify to the PVM daemon that a message is complete and should be placed into the buffer which will be read by the node program "N_net".

Corresponding to this host message is the node program code which removes these values from the buffer. See Figure 7-7. In this example each machine (node) in the cluster is already executing a program called "N_net". In this program is the code found in Figure 7-7. The message type is first set and a receive message command is issued to obtain the corresponding message from the PVM buffer. Upon completing this, the same values sent by the host in Figure 7-6 are read in by the node as shown by the code in Figure 7-7. It should be pointed out that the transmission in Figure 7-6 was done in a loop (with 'i' as the variable of index) such that each machine in the cluster will receive a set of values.

Node Program

```

/***** Receive NETWORK INFO From Host *****/
msgtype = 1;
rcv(msgtype);
getnint(&procs,1); /* No. of Processors*/
getnint(&no_layers,1); /* No. of Layers*/
getnfloat(&mu,1); /* Learning Rate*/
getnfloat(&inp_len,1); /* Input Vector File Length*/
getnfloat(&des_len,1); /* Desired Vector File Length */
getnint(&runs,1); /* Batch Runs */

for(i=0;i<=no_layers;i++){ /* PEs Per Layer*/
    getnint(&layer[i],1);}

```

Figure 7-7. Receiving of Network Information by the Nodes

In summary, all PVM communication from host to node or node to host is performed with a set of simple 'C' like syntax commands as those found in Figures 7-6 and 7-7. The PVM buffer previously mentioned is created and handled via software (pvmd daemon). Synchronization of the systems (nodes) is handled by machines waiting for the buffer to be full. For example in the case of retrieval, once all the nodes are given the network parameters and partitioned PE indices, they immediately begin computing the dot-product between the weights and input vector. Upon generating partial sums for each PE in the layer, the values are sent to the host for re-distribution to the nodes. A node will then sit idle until it has received the partial sums from the host which it will combine to form a completed partial sum. The synchronization therefore occurs in a machine waiting for intermediary results to be placed in the PVM buffer. Thus in the case of this neural application the synchronization was inherent in the program structure and the author did not have to be concerned with this issue. (It was taken care of automatically.)

CHAPTER 8 MULTIPLE DSP SYSTEM AND CLUSTERING RESULTS

8.1 Introduction

Chapter 8 presents the results of partitioning a multi-layer perceptron onto a system comprised of four DSPs and a cluster of four NeXT machines. The four DSP system was a master's degree hardware project developed under the direction of Dr. Michel Lynch. A NeXT cube prototype board containing a NeXT bus interface chip (NBIC), was used to mount four Motorola DSP 56000s and 64K of local SRAM per DSP [46]. This system was used to test the computational time models developed in sections 5.4 and 6.5. The author would have preferred to use the CNEL Machine however it was still undergoing hardware construction.

The cluster of NeXT machines consisted of two machines from the CNE lab and two machines from professor's offices down the hall. While the cluster could have been increased to contain as many as 9 NeXTs, only four were used for testing. This number was selected in order to draw comparisons to the four DSP 56000 based system. In either architecture (NeXT cluster or multiple DSP system), the partitioning scheme employed was the partitioning of the weights according to backpropagation alignment described in section 3.5.1.

Predictions for computation times were generated for each system and for various size networks. The models used in the predictions are those described in sections 5.4 and 6.4. The general model was given in section 5.4 and later adapted in 6.4 for the *common bus configuration*. Both the multiple DSP system and NeXT cluster can be considered a common bus configuration for the following reason. In either system, the communication

between processors was performed in a manner consistent with the common bus architecture. Beginning with the multiple DSP board, the host NeXT Machine (68040) was used to receive and transmit results between DSPs. For example, once intermediate results were generated by the DSPs, the host would be interrupted and sent the information (one DSP's group of values at a time). Upon receiving all the results, the host would then re-distribute the sums to the proper DSP. Similarly, one of the NeXT machines in the cluster of four acted as a host machine. All intermediate results were passed via Ethernet to this host NeXT and again redistributed by the host to the appropriate node NeXT machine. Thus in both cases, the host machine acted as a block of globally shared memory. In summary, having selected this model, it was possible to estimate the total computation time for a network running on each system, the corresponding local processing time portion and the time lost in global communication.

8.2 Estimating System Primitives

Before a prediction can be made, times must be estimated for the primitives associated with the models in sections 5.4 and 6.5. In the case of a floating point DSP based system, one method is to simply consult the technical literature and obtain multiply accumulate and transfer times for the processor and memory. This can also be performed for other types of processor instructions and global memory transfer rates. This yields a good starting estimation of how fast your system will run. However this may lack accuracy in the case of a fixed point DSP based system or possible in the case of workstation clustering. In the fixed point multiple DSP system, a substantial amount of processing may go towards scaling data which is not accounted for in the computation models given in (5-8) through (5-14). Thus the actual computing times will be far worse than the predicted times. With a cluster of workstations, the problem worsens. e.g., How does one obtain times for transferring information to/from the processor? Technical specifications are

usually not available at this level. Another problem is how to take into account the inefficiencies created by the compiler. e.g., extra assembly code created by compiling a high level language. For example, in the case where an application has been written in 'C' and compiled to execute on a particular machine, the computation time will be normally greater than hand coding the application in machine code. In addition, time may also be lost in indexing variables in memory. Unfortunately none of this is taken into account by the models given in sections 5.4 and 6.5. Because of this, a hybrid method of grouping primitives and estimating group (subroutine) times is presented.

This method consists of estimating an operation's computation time as follows. The actual application (neural net program) is written such that modules exist which correspond to the equations (5-8) through (5-14) and (6-4). Having written and debugged these modules, they are tested and timed for different size networks. For example, in the computer cluster, subroutines were written to perform the basic MLP functions such as dot-product, nonlinear transform, error propagation and so forth. Once found to be error free, each routine was timed using a C language time structure for 10 different size networks. Upon measuring the computation time for a module, it was divided by the number associated with the operation count. For example in the dot product, the count corresponds to N_k times N_{k-1} which is equal to the number of PEs or inputs in the $k-1$ and k layers. The nonlinear transform count would be N_k . The exact count for a particular module is therefore obtained by looking at the appropriate operation or phase in the computation time equation. See (5-9) through (5-16) and (6-4). *In summary, the module's execution time is divided by the operation count such that a primitive is generated for a particular operation (routine).* The local processing modules thus correspond directly with the phases in the computation time equation (5-14). Similarly, counts associated with modules written for transmitting partial nets and/or error terms are obtained from equation (6-4).

Table 8-1. Multi-layer Perceptron Routines and Operation Counts

<u>Routine Description</u>	<u>Associated ATPD Equation</u>	<u>Count</u>
dot product	(5-8)	$N_k N_{k-1} / P$
nonlinear transform	(5-8)	N_k / P
output error	(5-9)	N_k / P
back error thru PEs	(5-10)	N_k / P
back error thru wts	(5-11)	$N_k N_{k-1} / P$
delta wts	(5-12)	$N_k N_{k-1} / P$
wt update	(5-13)	$N_k N_{k-1} / P$
send partial sums (retrieval) to host	(6-4)	$(P-1)N_k$
get partial sums (retrieval) from host	(6-4)	$(P-1)N_k$
send errors (backpropagation) to host	(6-4)	N_k
get errors (backpropagation) from host	(6-4)	$(P-1)N_k$

The advantages of estimating timing primitives at a module level is that it simplifies the models given in section 5.4 and takes into account all the potential inefficiencies generated by a user's application code. In the case of the multiple DSP system, indexing, scaling and other elements not present in the ATPD timing models were taken into account by timing an actual application operations. The counts associated with these operations were obtained by examining (5-8) through (5-13), (6-4) and matching it to a particular operation or routine. See Table 8-1. A list of subroutines and there associated count numbers is given. In Table 8-1, all local operations are divided by the number of processors P due to the even distribution of weights, PEs, inputs, outputs and processing time. N is the

number of PEs or inputs in a given k layer. Thus the counts for a routine depend on the instance of the routine (which layer of the net it is operating on).

The last four communication times in Table 8-1 are associated with receiving/transmitting results globally to/from the host. This is not a local operation and therefore is not divided by P . A specific explanation of how these counts are obtained for global communication is given in section 6.4.

In summary, primitives can be estimated from the actual code which executes on the parallel system. This is done by running a subroutine several times, measuring the execution time, computing an average and then dividing it by an associated count. This can be performed on any type of system which contains a clock or timing mechanism. In the case of the multiple DSP system, an external 16 bit timer chip was interfaced to one of the DSPs for measuring the speed of a particular routine.

8.3 Multi-layer Perceptron Results

A multi-layer perceptron was first implemented on a cluster of NeXT machines and then intermediate results (activations, error values, etc...) were printed to test and debug the code running on the four DSP system. Because the partitioning was identical for the two different systems (workstation cluster and multiple DSPs), the results obtained from the C code modules written for the cluster could be compared against the results obtained from the DSP-based system. This turned out to be an extremely fast and efficient method for testing the DSP code. A general purpose MLP application was written in two months and can handle nets up to 240,000 weights, in almost any combination of PEs per layer and number of layers.

In addition to creating intermediate results for checking DSP code, module (subroutine) primitives were estimated by executing various size networks, timing them and computing average values based on counts and multiple runs. Refer back to section 8.1 for

further details. The base test network was a simple 48 input, 8 hidden layer PEs and 12 output layer PEs static neural network. The 48 inputs corresponded to spectral information resulting from passing a window of time domain signal through an FFT. The time domain signal was originally obtained by sampling a single guitar note. The 12 outputs of the net corresponded to classifying 1 of 12 guitar notes. Thus the network was trained to recognize a guitar note (by using its frequency information) out of a possible group of twelve. The network results show that the network did in fact train but will not be presented in this document. Instead emphasis will be placed on the computation time associated with training a particular size network.

Upon training the 48-8-12 base network and obtaining the estimations for subroutine timing primitives, the network was scaled up by 5 and the process repeated. Thus primitives were obtained for the case where the network consisted of 240 inputs, 40 hidden PEs and 60 outputs. In addition, primitives were obtained for networks of size 480-80-120, 960-160-120 and 2400-400-600. By scaling up the original network and obtaining new primitives, it was possible to estimate primitives which are dependent on the number of operations that will be performed (e.g., the count described in section 8.1). For example, given the 48-8-12 size network, each processor will have to compute 2 nonlinear transforms (execute the nlt loop twice) for the activations associated with the hidden layer. In the case of the 240-40-60 size network the number of nonlinear transforms is 40/4 or 10. Thus by scaling the network up by 5, the count associated with the nlt in the hidden layer also increases 5 times. This also holds true for scaling the base network up to the 480-80-120, 960-160-240 and 2400-400-600 size networks. *The end result is that a primitive can be obtained for an instance of code (loop or subroutine) which is based on the count or number of instances of that function.* For example, we can determine primitives for an operations such as the time to communicate a single datum to the host based on how

many values will be sent to the host. The primitives now become functions of the size of the network under study.

8.3.1 NeXT Cluster

Having obtained primitives for the base network and four scaled up networks, linear interpolation was used to yield a primitive for a network in-between the five test cases. Thus we begin with a set of golden primitives (based on 5 different size networks) and use linear interpolation to generate new primitives for estimating computation times for networks different than the original five. Other size networks tested were those of configuration 96-16-24, 192-32-48, 384-64-96, 576-96-144, 720-120-180 and 1008-168-252.

Actual computations times for all 11 different size networks for the cluster are shown in Figure 8-1. The network sizes are given in terms of total number of weights (horizontal axis) and are presented in a log scale. Figure 8-1 illustrates the amount of time required to update all the weights once for a given network. This is termed one complete learning iteration. The cluster consisted of four NeXTs and the actual shown in Figure 8-1 is the greatest time of the four processors (worst case) executing.

In Figure 8-1, total time has been broken out into local processing time and global communication time. The communication time is shown to be nearly constant with increasing network size while local processing grows very rapidly. This is due to the efficient partitioning of the network according to backpropagation alignment. As mentioned in section 3.5, local processing was expected to grow with respect to N^2 while global communication grows with respect to N .

Another important point to observe are the communication values in the range of 10^5 to 10^6 network weights. A slight dip and then a rapid increase in communication time can be found around the 2×10^5 weight point. This is due to how well the transmitted data fits an optimum Ethernet packet (message). An Ethernet packet contains a certain amount

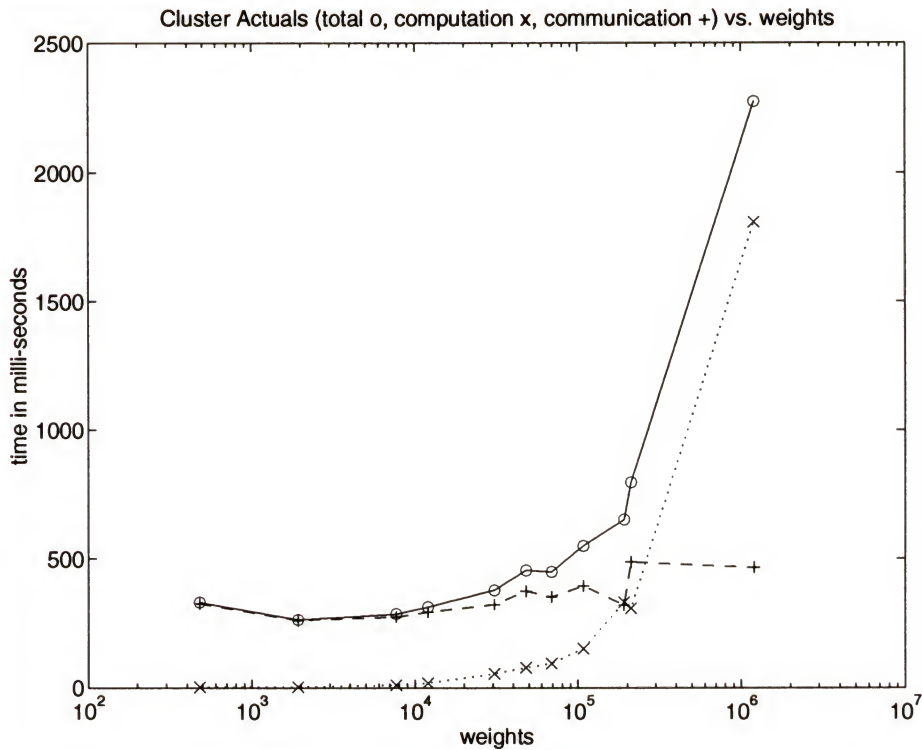


Figure 8-1. MLP Cluster Computation Time

of header (overhead) associated with address, message type and data verification in addition to the actual data sent to another machine. When the global communication data fits the data size allowed in the packet, the communication time is most efficient (lowest). On the other hand, if a small portion of data spills over into another packet it may be smaller or nearly the same size as the header. Hence the majority of your message is lost in header overhead and Ethernet communication times worsen.

8.3.2 Multi-DSP System

Computation times were also measured for the multiple DSP system and are shown in Fig. 8-2. In the cluster case, ten points per curve were generated that correspond

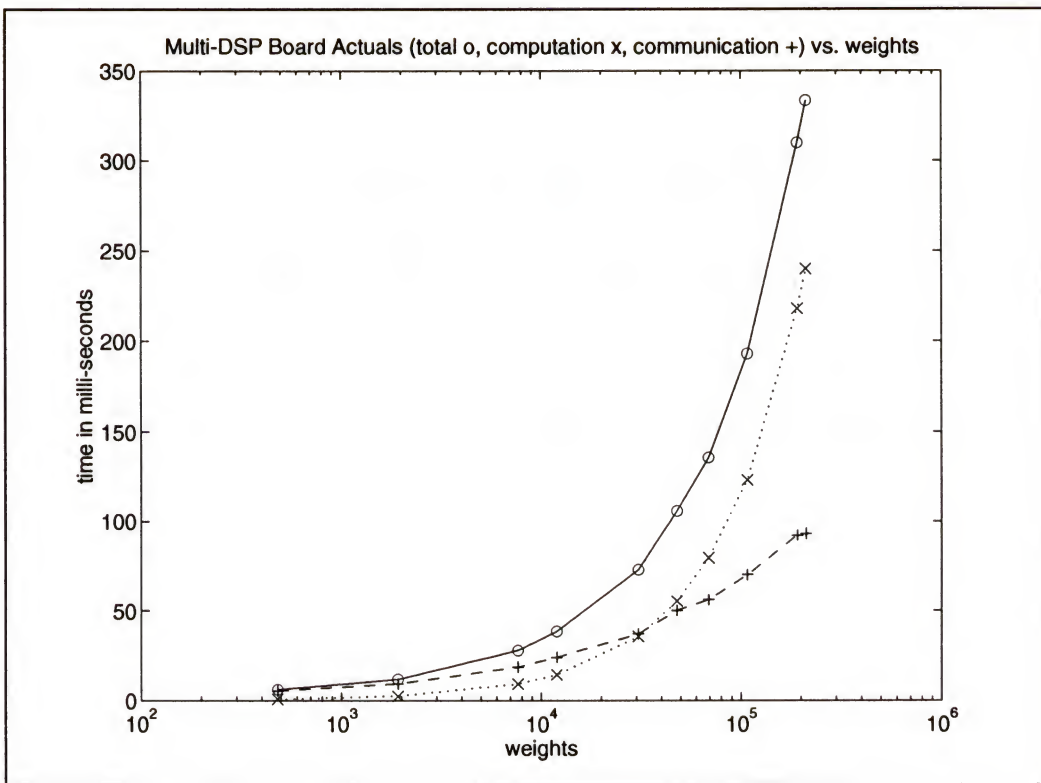


Figure 8-2. MLP Multiple DSP Board Computation Time

to networks of size 48-8-12, 96-16-24, 192-32-48, 240-40-60, 384-64-96, 480-80-120, 576-96-144, 720-120-180, 960-160-240 and 1008-168-252. The eleventh largest net (2400-400-600) was too large for the DSP memories and thus could not be tested. This network consists of 1,200,000 weights which far exceeds the 256K words of memory on the four DSP board (64K per DSP).

Returning to Figure 8-2, the local processing curve is found to rise rapidly with increasing network size as in the cluster example. However one difference between Figures 8-2 and 8-1 is that the communication time in 8-2 does not level off. This is due to the following reason. In the cluster example, as the number of globally communicated values

increases, the communication primitive (Ethernet communication time) decreases. The more data sent across Ethernet, the more efficient it becomes. However in the DSP case, the communication primitive does not shrink with increasing message size. The primitive was found to essentially remain constant regardless of network size. Thus as network size increases Figure 8-2, the amount of information globally communicated increases and therefore the time associated with sending these values must also increase. In the multiple DSP system the communication between DSPs is performed by the host NeXT machine via a 68040 bus transfer from one local DSP memory to another local DSP memory. Hence this operation has no header and very little overhead associated with it. Because of this, the primitive associated with sending a small quantity of information is nearly the same as the primitive associated with sending a large quantity of information. A larger number of values transmitted between DSPs will therefore always take more time to complete than transferring a smaller set of values. This is not always true for the Ethernet communication.

8.3.3 Prediction Accuracy

Figure 8-3 illustrates the prediction error for both the DSP group and computer cluster. The two curves were obtained in the following manner. Total computation time estimates (for a processor in the parallel system) were made for the networks mentioned in sections 8.3.1 and 8.3.2. This was eleven different size networks for the cluster and ten for the multiple DSP system. The total computation time estimates were generated according to the procedure given in section 8.2. The estimate represents the amount of time required to perform one learning iteration (updating all the network weights one time). Having obtained this estimated value, it was subtracted from the actual with the result then being divided by the actual. In the case of the DSP based system, the actuals were always greater than the estimated times. In the cluster, 10 of the 11 actual computation times were less

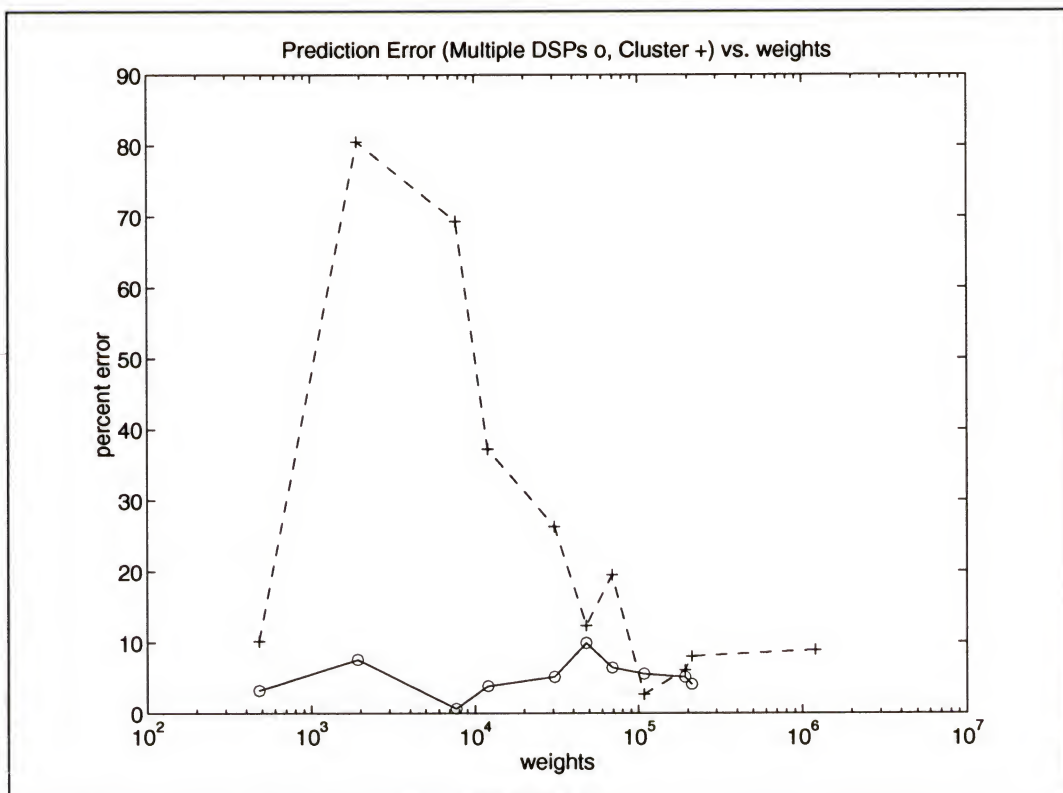


Figure 8-3. Cluster and Multiple DSP System Prediction Error

than the predicted times. The point being that the percent errors can be read directly off of Figure 8-3 and are not plus or minus the percent read off of the figure.

In summary, all the DSP estimates are within 10% of the actual values. This is due to a more consistent DSP processing time when compared to the cluster and the fact that the global communication is a DSP bus transfer rather than an Ethernet transfer. The cluster predictions however do reach the same level of estimation error when the network is 10^5 weights or greater. This is due to two main reasons. The first is that the component of local processing time (which is predictable) becomes equal to and finally greater than the Ethernet communication time. See Fig. 8-1. The other is that the Ethernet communication

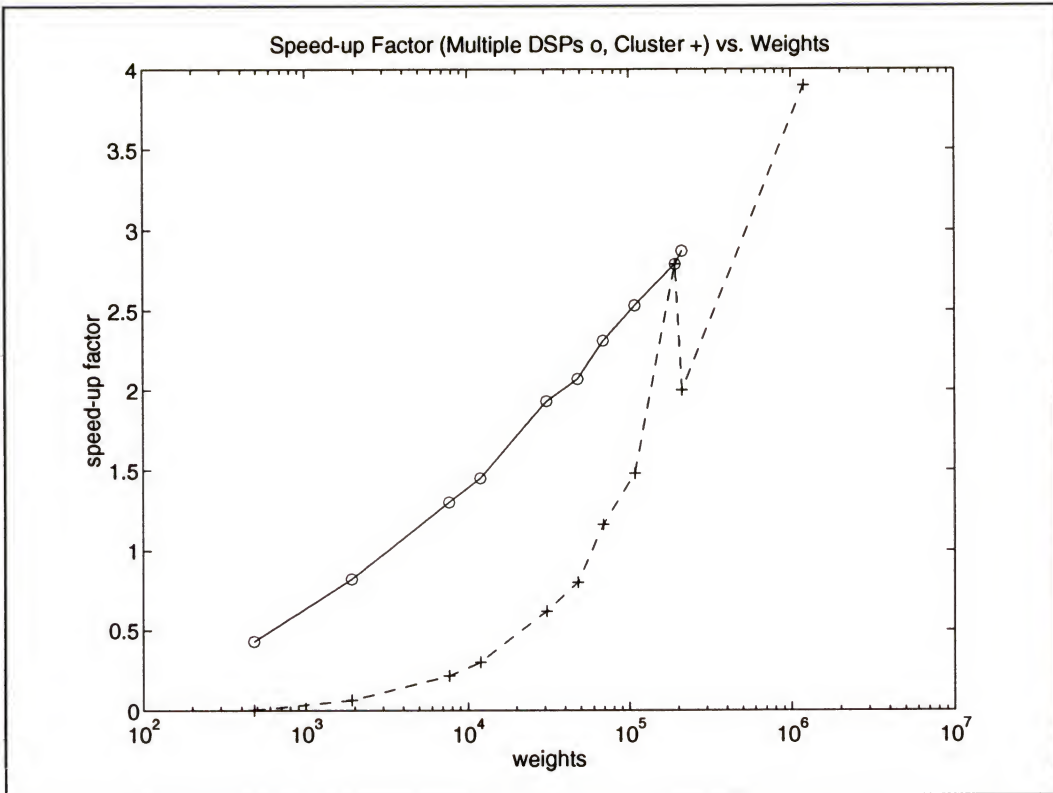


Figure 8-4. Cluster and Multiple DSP Speed-up Factors

time becomes more consistent (from a primitive point of view) when large quantities of data are transferred.

8.3.4 Partitioning Efficiency of the Workstation Cluster and Multiple DSP Board

The efficiency of the backpropagation alignment partitioning scheme will now be analyzed for the cluster and multiple DSP board. One of the best ways to observe this is by looking at the speed-up factor of the parallel system versus the serial machine. See Figure 8-4. The Total Computation times found in Figures 8-1 and 8-2 are now plotted versus the amount of time required by a single processor to complete one learning iteration. In the multiple DSP board, the serial machine is a single DSP56000 and in the cluster it is one

non-turbo NeXT Machine. The weights again correspond to the networks mentioned in sections 8.3.1 and 8.3.2 (eleven for the cluster and ten for the DSP based system).

Several interesting observations can be made from Figure 8-4. The first is that for small networks (less than 4×10^3 weights) neither system shows a speed-up greater than one. In both cases, the inefficiency can be attributed to the large amount of global communication time versus local processing time. See Figures 8-1 and 8-2 in the range of 10^3 - 10^4 weights. When compared against the DSP based system, the cluster is extremely inefficient in this range. This is due to the Ethernet transfer being much slower than a Motorola 68040 bus transfer in the multiple DSP system.

Another major observation is that the DSP system's curve in Figure 8-4 appears nearly linear when compared to the cluster. Again this is due to the differences in communication. The DSP communication has a near constant transfer primitive. Meaning that if 100 or 1000 values are transferred between DSPs, the unit transfer time (e.g., time to send one item) is the same for both cases. However as mentioned earlier, the cluster global transfer primitives are not constant. The unit time shrinks with respect to increasing amount of data transferred between workstations.

One final observation is that for large networks, the DSP board's highest speed-up is expected to be at most three. The curve in 8-4 cannot be extend much further than that shown. The last network tested was 1008 inputs, 168 hidden PEs, 252 output PEs which consists of 211,680 weights. This is very close to the 256Kword limit of the multiple DSP board. The cluster on the other hand is limited by DRAM memory which is 8M bytes per machine. Assuming that 80% of this is available and that the weights are stored as floating point numbers (4 bytes per weight), the maximum net allowed is 6,400,000 weights or 6.4×10^6 . If we extend the cluster curve found in Figure 8-4, the speed-up factor is expected to be greater than four due to the faster processors comprising the cluster when

compared to the non-turbo serial machine. For example, the serial processor was chosen as a non-turbo NeXT and two of the four cluster processors were turbo NeXTs which are approximately 1.3 times faster than a non-turbo. Thus the maximum speed-up is expected to be around 4.6. The author chose the serial machine as a non-turbo because this was the type of machine available in the CNE Lab. The two turbo machines however were not resident in the lab (professor's offices) and thus were only available as remote systems. Hence lab members almost always use the non-turbo machines found in the lab and thus they were used for comparison.

8.3.5 Comparing the Workstation Cluster vs. the Multiple DSP Board

Figure 8-5 illustrates how the cluster performs versus the multiple DSP board. The Cluster Total Time from Figure 8-1 was divided by the multiple DSP system's Total Time (Figure 8-2) for the 10 test networks. The value appears to converge around 2.39×10^5 weights. As expected, the DSP system greatly out performs the cluster for smaller networks. However for large networks, the cluster performance is very near that of the multiple DSP system. We attribute this to the Motorola 68040's fast floating point multiply, internal cache and pipelined architecture. In the DSP56000, fixed point numbers must be constantly scaled before and after a single dot product multiplication. This greatly reduces the overall efficiency of the dot product when compared to the non-scaling floating point operation in the 68040.

8.4 Focused Gamma Net Results

In addition to the multi-layer perceptron, a focused gamma net application was written and tested on the four NeXT machine cluster. The focused gamma net was partitioned as discussed in section 4.5 and the computation time estimator was updated to include this new topology. In the focused gamma net application, two new routines were written for computing gamma kernel outputs (kernel_out) and gamma parameter updates

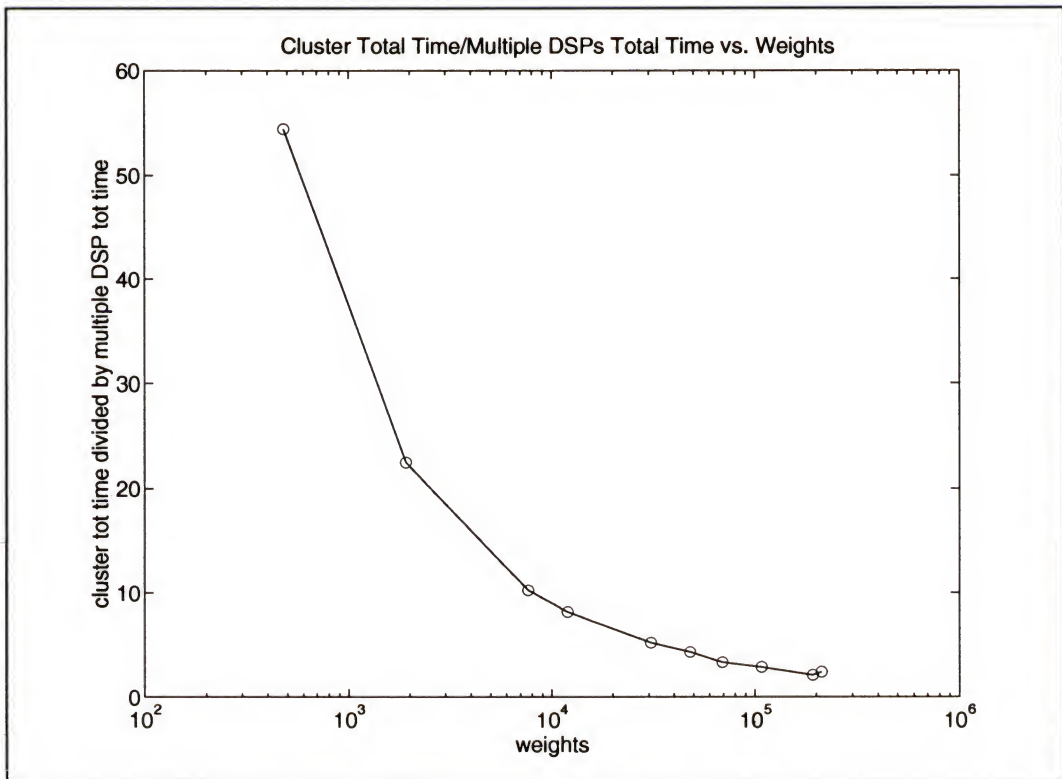


Figure 8-5. NeXT Cluster vs. Multiple DSP System

(gamma_update). Otherwise, existing routines were adapted to include the additional focused gamma net processing. For example, the MLP dot product routine used in retrieval was expanded to include the dot product calculation between the kernel outputs and the gamma weights. Similarly, the MLP weight update subroutine was adapted to include updating the gamma weights and backpropagation of the error through the gamma weights was added to the original MLP backpropagation routine. Because of this, only two new primitives were required. The new primitives are those associated with the kernel_out and gamma_update routines.

In the case where the gamma processing was added to an existing routine, the original MLP primitives were used. Because the computation structure did not change between the MLP the focused gamma net, the only difference between the two topologies was in the count value. For example, in a two layer focused gamma net the number of dot product computations (count) in the first layer is three times that for a similar MLP realization. The operation (e.g., dot product process) however is the same in both cases and thus the original primitive was employed for the focused gamma net estimator.

In the case where two new routines were written (e.g., two new types of computation), primitives were estimated as discussed in section 8.2. The actual application routines were timed for networks of structure 48-8-12, 240-40-60, and 480-80-120 with two gamma kernels per input in each network. Recalling from section 8.2 that a fifth network, (2400-400-600) was used to estimate timing primitive values, this could not be tested in the case of the focused gamma net. This was due to the following problem. The 2400-400-600 focused gamma net with two kernels per input contains 3,120,000 weights. Because each weight is stored as a floating number this equates to over 12M bytes. When divided across the four machines, this presented a problem on the turbo machines (Andrew and Barney) located in professor's offices. Andrew and Barney contained only 8M of DRAM while the two non-turbo machines in the CNE Lab were configured with 16M of DRAM. Because of this smaller quantity of DRAM and the increased weight requirement of the focused gamma net, not all of the weights could be stored in DRAM. Thus weights were moved to swap disk space on the hard disk by the operating system. This then caused the local processing on the 8M machines to be 100 times slower than the 16M DRAM machines. Because of this and the desire to use the same machines as those used in the MLP tests, the 2400-400-600-2 gamma layer focused gamma net result was discarded.

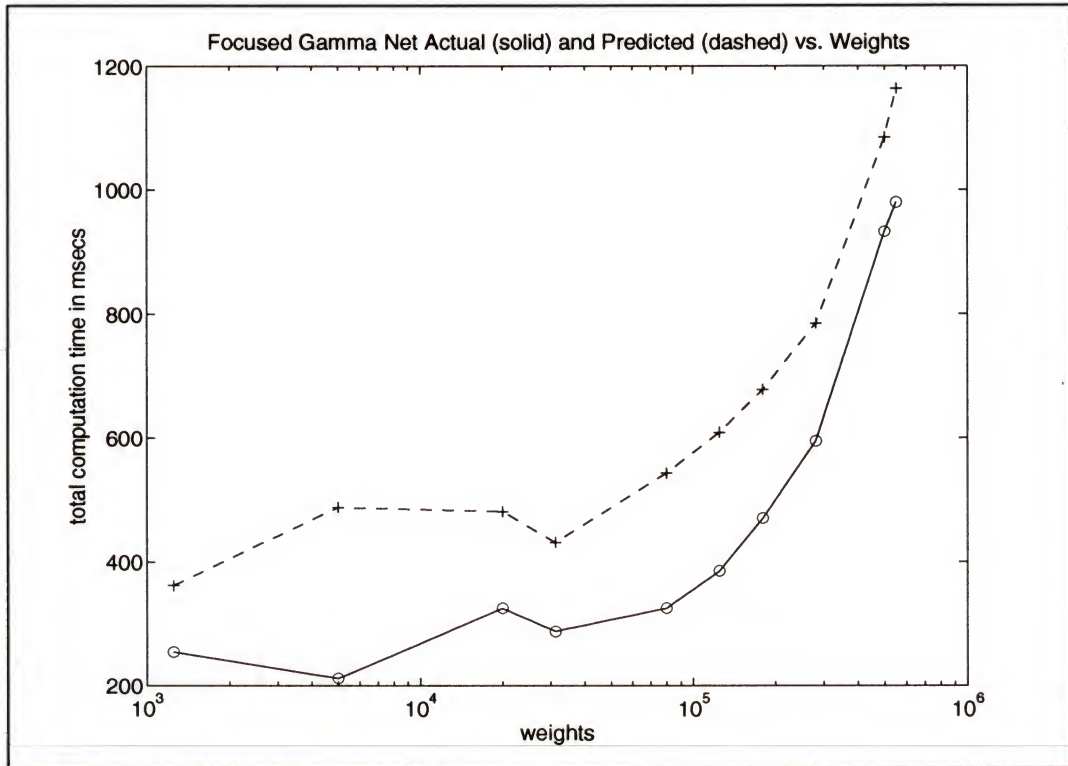


Figure 8-6. NeXT Cluster, Focused Gamma Net Actual vs. Predicted

All future tests mentioned in this chapter will consist of only 10 of the 11 networks given in the section 8.3.1. These are the same network sizes tested on the multiple DSP system.

Upon obtaining the new primitives and adapting the MLP estimator to include the focused gamma net, several estimations were generated and tested. The results of this are shown in Figure 8-6. Computation Times for performing one learning iteration (single update of all weights) were estimated and compared to actuals for networks 48-8-12, 96-16-24, 192-32-48, 240-40-60, 384-64-95, 480-80-120, 576-96-144, 720-120-180, 960-160-240 and 1008-168-252. Results are similar to those found in the MLP cluster predictions and tests. Predicting total computational times for smaller networks is prone to error.

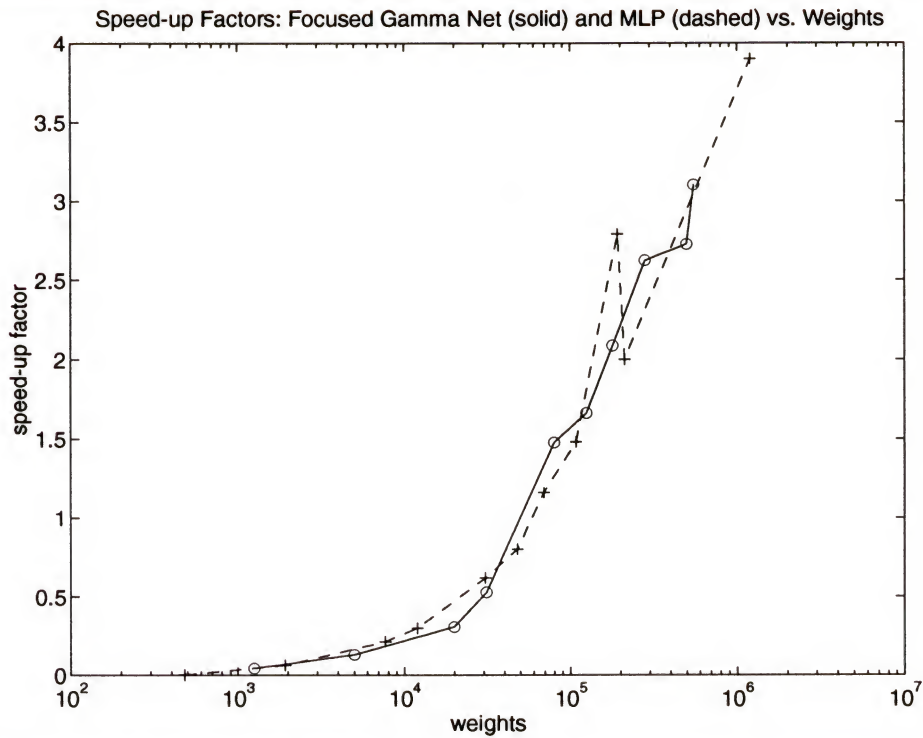


Figure 8-7. NeXT Cluster Speed-up, Focused Gamma Net Actual vs. MLP

This is due to the large portion of global communication time and the non-deterministic nature of the Ethernet communication link.

However for larger networks, the difference between the predicted value and actual is a smaller when compared to the total computation time. This is due to the larger portion of local processing time which is more predictable than Ethernet communication times. Checking the results, it was found that local processing time can be estimated to within 90% accuracy for networks greater than 10^4 weights. Methods for improving the global communication times will be presented in the conclusion of this dissertation.

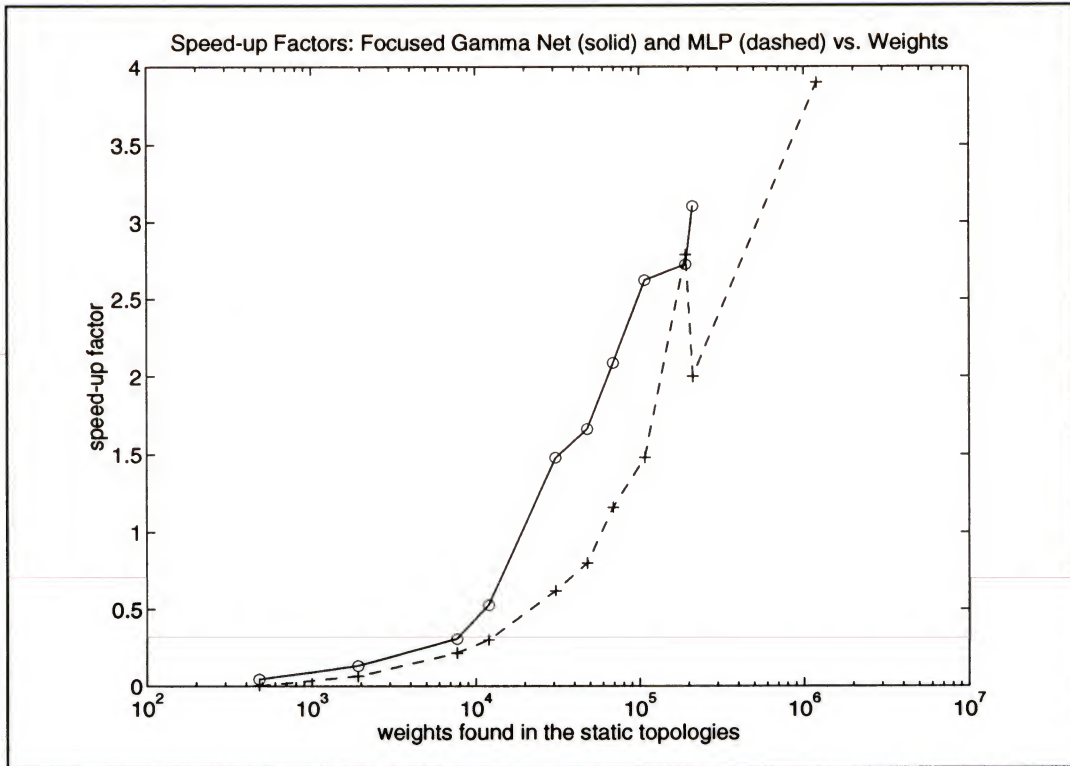


Figure 8-8. Focused Gamma Net Actual vs. MLP Speed-up (common weights)

The speed-up of the cluster executing a focused gamma net was also compared against the speed-up factors computed while executing a multi-layer perceptron. This is shown in Figure 8-7. The speed-up was computed for each curve by dividing the single NeXT (non-turbo) total computation time by the cluster's worst processor total computation time. Returning to Figure 8-7, the focused gamma net curve is shown to follow the MLP very closely. What this concludes is that the speed-up factor is based on the number of weights in the network and not whether the network is a MLP or focused gamma net. However one must remember that given the same number of input, hidden layer PEs and outputs, the focused gamma net will contain more weights. For example, in the 48 input, 8

hidden PE and 12 output base network there are $48 \times 8 + 8 \times 12$ weights in the MLP. In the focused gamma net where there are 2 kernels per input there are $3 \times 48 \times 8 + 8 \times 12$ weights. Thus if we plot the speed-up factors for the two algorithms against the weights that are found in the static topologies of both nets, the focused gamma net is more efficient. See Figure 8-8. This is a fair comparison because normally when one experiments with both types of algorithms, the number of inputs, hidden PEs and output classifications is kept the same in both cases. In other words, one only adds the kernel memory structure on top of the static network topology when moving from the MLP to the focused gamma net.

In summary, Figure 8-8 presents an important point when comparing the focused gamma net and MLP computation times. In a serial system, the focused gamma net will always require additional computation time when compared to a MLP containing a similar base topology. However in the parallel system, the differences may not be as significant. The focused gamma net can be partitioned more efficiently across a parallel system than the MLP.

8.5 Chapter Summary

As mentioned in section 8.3, both the multiple DSP system and NeXT workstation cluster were found to conform to the common bus model developed earlier in Chapter 6. The global communication overhead found in each system matched the transfer of partial nets and errors to/from the global memory block of the common bus architecture. Therefore the model given by equations (5-9) through (5-16) and (6-4) was selected to yield total computation time predictions. Having selected this model, estimations were then required to fill in the necessary system primitive variables. Depending on the software level and availability of hardware, two primitive estimation methods were presented.

The first was to estimate the primitives via specifications from the hardware manufacturers (e.g., base processor specs, memory timing diagrams, bus cycle timings, etc.)

Essentially, the user estimates (from paper studies) the global and local memory transfer times, internal processor transfer times, processor multiply-accumulate and nonlinear transform times and then directly inserts them into the total computational time equation. This method is highly dependent on the ability to obtain clear concise information from the hardware manufacturer.

The second method was to create software routines which corresponded to phases in the computation time equation and then time them individually. This had several advantages in that overhead created by the application in the form of indexing, scaling or compiler inefficiencies (how well the high level language translates to machine code) could be taken into account in the prediction model. The downside was that this required both the architecture and application software to be functional. For architectures which have not yet been constructed (e.g., CNEL Machine discussed in Chapter 6), only the paper study primitive estimation method is available.

Because both the multiple DSP56000 and NeXT cluster parallel systems were functional, the subroutine timing method was employed. Routines given in table 8-1 were executed at least 30 times and averaged for a given size network. From these averages and the associated operation counts, routine or module primitives were generated which corresponded to the amount of time to perform a single operation. e.g., single dot product, nonlinear transform, subtract two values, send/receive one value, etc... Having generated primitives for a base size network (48-8-12), four other networks were run for primitive estimations. This then produced five primitives values for each particular module or operation. *The purpose was to model how an operation primitive would change according to the number of times it was executed (count value).* Given these five primitives for each operation, linear interpolation was used to obtain the value when the count differed from

that produced by the five test networks. This enabled the predicting software to span a large range of available network configurations.

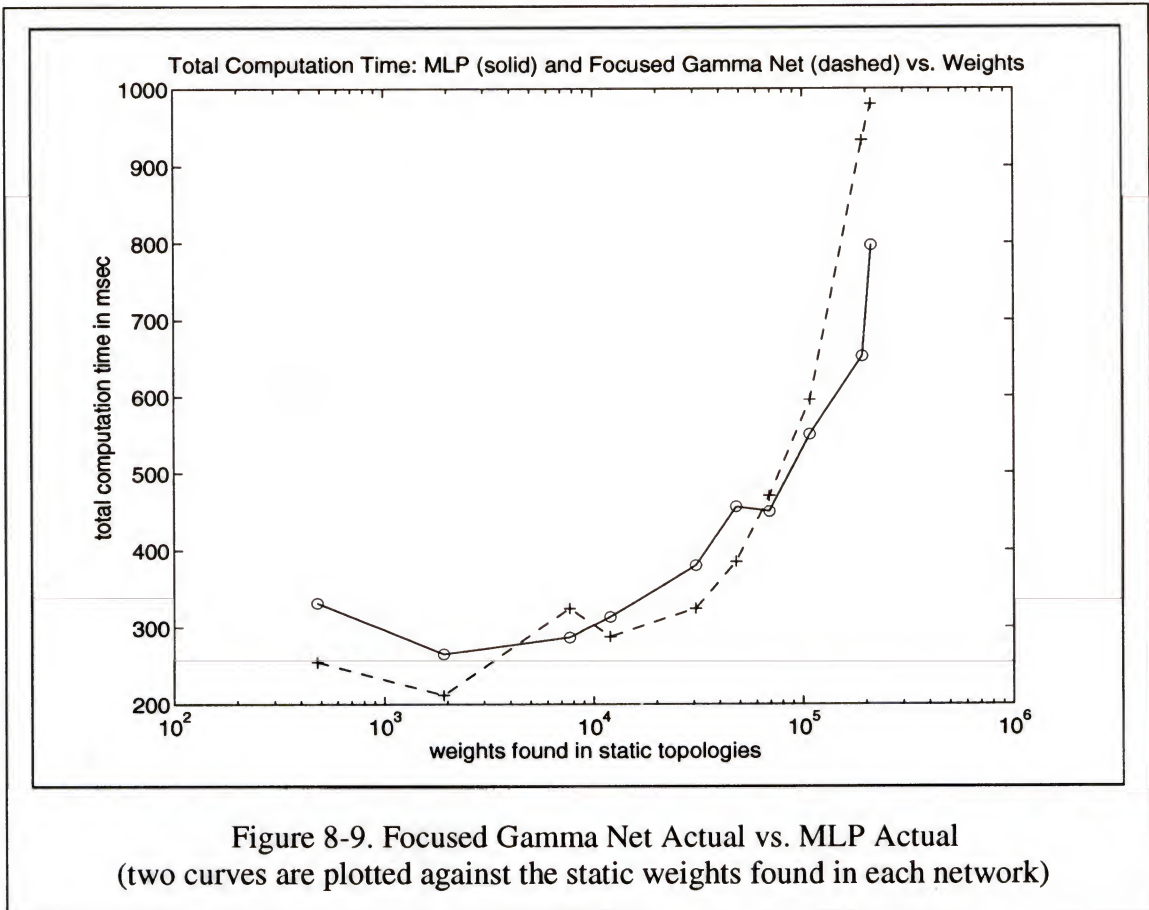
Once the total computation time primitives were obtained, predictions were made for 5 different networks (6 in the cluster case) executing multi-layer perceptron learning and compared to the actual total computation times. The predictions for the multiple DSP system were found to be within 90% accuracy. The cluster predictions however were found to be inaccurate for smaller nets (10^3 - 10^4 weights) but approached the accuracy of the DSP predictions for larger networks (10^5 weights and above). See Figure 8-3. The differences between the accuracies of prediction for the two systems was attributed to the ability to predict the global communication time. In the DSP based system, the global communication was based on a 68040 bus cycle which has a low variance (5%). In the cluster however, the global communication is performed via an Ethernet line which is considered non-deterministic. The cluster predictions only become reasonably accurate when the portion of local processing time is greater than the global communication in the total computation time. See Figures 8-1 and 8-3.

Upon checking the accuracy of the predicting software, the multiple DSP system was then compared to the cluster for efficiency in parallelism and computing performance. The DSP based system was found to have a higher degree of parallelism across the 10 test networks when compared to the cluster. See Figure 8-4. This was attributed to the ratio of communication time versus local processing time for each of the two systems. In the multiple DSP system the ratio was found to be far less than the cluster's ratio. Again the difference between a Ethernet and processor bus transfer is given as the reason for differences in the two speed-up curves in Figure 8-4. An interesting observation though is that the two curves begin to converge for networks with more than 10^5 weights. In this range, Ethernet communication becomes more efficient and less a contributing factor to

the overall computation time. This in turn improves the performance of the cluster and makes it comparable to the multiple DSP system. See Figure 8-5. The performance of the DSP56000 is only twice that of the NeXT Machine for the largest network. This is due to the scaling overhead incurred by the DSP56000 and the fast multiply accumulate operation in the 68040.

Upon running the multi-layer perceptron learning tests on the cluster, the focused gamma net was implemented. This required two new routines and minor additions to be made to existing C routines. The two new subroutines were timed as in the MLP case and corresponding module primitives were generated. No modifications needed to be made to the primitives associated with the adjusted routines. The operation stayed essentially the same; only the count increased. After obtaining the two module primitives, the estimation software was updated and predictions were again made for 10 networks. Comparing these values to the actuals, the results are very similar to those found in the MLP case. See Figure 8-6. Again the predicted curve is shown to converge on the actual as the number of network weights increases. The reasons for this are the same as those found in the MLP case. Shrinking Ethernet communication time and a larger portion of local processing time comprising the total computation time number.

The speed-up or efficiency of parallelism was also found to be very similar to that found in the MLP testing. See Figure 8-7. The additional time created by the two new gamma routines appears to be negligible. However when looked at from an application point of view an interesting result arises. Whether one uses a MLP or focused gamma net in an application, the input, hidden layer units and outputs remain the same for both networks. However the number of weights in the input layer increases $G+1$ times for the focused gamma net (where G is the number of gamma layer weights.) This then increases the amount of local processing but has no effect on the global communication. As a result,



the parallel speed-up increases when compared against the MLP realization. See Figure 8-8. This brings up an interesting observation. While additional time will be required for serially computing focused gamma net learning, it may not be the case on the parallel machine when compared with a similar MLP realization. It will depend on the size of the network and the ratio of global communication time versus local processing time. This then lead the author to wonder how much more computation time was required to train the focused gamma net when compared to the MLP. See Figure 8-9. For smaller nets, there is no significant difference between total computation times. This is due to the large portion of global communication times in both signals. However for larger nets the focused

gamma net has a larger total computation time as expected. However because of the larger number of weights in the focused gamma net, it was originally expected to be far worse than that shown in Figure 8-9. For example in the last two networks tested, the number of weights found in the MLP static topology is 192,000 (960-160-240 configuration) and 211,680 (1008-168-252 configuration). Similarly, the number of weights in the static topology section of the focused gamma net is also 192,000 and 211,680. However the total number of weights in the focused gamma net (including the gamma weight layers) is 499,200 and 550,368 for the last two nets. Thus these figures are more than double the number of weights in the MLP and yet the differences in the two curves show only a 25-30% degradation in computing speed for the focused gamma model. The final point being that significantly increasing the amount of algorithm computation for the serial machine may not necessarily significantly degrade the performance of a parallel system if the added computation is in the form of local processing.

CHAPTER 9 SUMMARY AND FUTURE WORK

9.1 Summary of Research

In Chapter 1, the main goal of this research was introduced. This was to develop a method which would quantify the efficiency of a particular partitioning scheme on a given parallel architecture. Minor goals were then also proposed which included identifying the architectural design trade-offs for a given hardware/software realization and determining an efficient partitioning scheme for recurrent artificial neural networks.

With respect to the main goal, it was deemed that the analysis method is constrained both by the algorithms comprising the application and the implementing hardware. Therefore beginning with the algorithm, the multi-layer perceptron (MLP) was selected for analysis of parallelism in Chapter 2. This was performed by *decomposing the MLP backpropagation process into phases and then writing the equations found in each phase in a summation form*. In addition, a flow chart was presented to illustrate the dependencies of an individual phase. Later in Chapter 3, the dependency graph (DG) was introduced as another example for locating the parallelism in an algorithm [15]. While it is difficult to select an optimal method for observing parallelism in an algorithm, the important point is to find a process which works best for the individual researcher. Restated, the user should employ whatever means are necessary to intimately understand and locate all the parallelism inherent in the algorithm under study.

Having presented the MLP nomenclature and parallel analysis in Chapter 2, the focus of the dissertation shifted to previous neural network hardware implementations. This was done to observe the constraints imposed by different researcher's architectures

and the associated partitioning schemes. From this study, several conclusions were drawn. The first being that while analog implementations offer the best computational bandwidths, they lack precision in weight representation. The typical analog weight resolution was found to be 5-8 bits. This was considered a potential problem in the backpropagation learning process; the incremental weight changes (due to multiplying by the learning rate μ) may be much smaller than the 5-8 bit value assigned to the weights.

Another problem with analog neural systems is their ability to scale. If the application network doubles in size, additional hardware is required. The new network can not be re-mapped or partitioned onto the old system. This in turn limits the flexibility of the system. Further limitations in flexibility were also pointed out from a processor functionality point of view. The analog system normally contains a custom processing element designed for a few specific neural net functions. Therefore the system will not be able to compute any neural net pre-processing tasks. e.g., filtering, FFTs, etc... Because of these limitations, a digital realization was then considered for accelerated artificial neural processing.

The digital system was found to sacrifice speed for scale-ability and computational flexibility. This was considered a better trade-off for the CNE Lab. The lab required a system which could test numerous different network topologies and learning mechanisms. Thus two digital systems were analyzed in great detail, Nagoya Institute's Neuro Turbo and S.Y. Kung's systolic ring arrays [31][32]. From the Neuro Turbo analysis, emerged a partitioning scheme which could be adapted for other types of coarse grain parallel systems. This scheme was referred to as partitioning the weights according to backpropagation or backpropagation alignment for short. It evenly distributes the network inputs, weights, PE activation calculations and outputs across the P processors in the parallel system. Another benefit of this partitioning scheme was the order of growth for global communication when compared to local processing. Global communication was found to

increase with respect to N (the number of PEs in a layer) while local processing grows with respect to N^2 . This suggests that the parallel system will increase in efficiency (greater speed-up factor) for increasing network size.

After reviewing current neural accelerators, Chapter 4 shifted back to widening the scope of algorithms analyzed for partitioning. A dynamic neural network known as the focused gamma net was presented and again analyzed for parallelism. The focused gamma net contains a short term memory structure in the form of a locally recurrent delay line (gamma kernel) at the input of the net. The purpose of this is to allow the use of input signal information from a previous state(s). Thus enabling the network to better track or classify a time varying input. The depth of the memory is controlled by the gamma feedback parameter, γ , and the number of kernels (recurrent delays) stacked on top of each input element.

Having introduced this new network, a discussion proceeded as to how this algorithm could be partitioned across a coarse granular parallel system. In addition, the partitioning of the more general algorithm, the gamma model (where kernels can exist at any input or PE layer) was also discussed. In both cases, the partitioning consisted of applying the backpropagation alignment scheme to each layer of weights in the network. Basically the network weight layers were partitioned across the P system processors in the same manner as the layer of weights in the MLP. gamma kernel structures were then partitioned such they remained resident with the partitioned inputs or PEs. In other words, a single gamma structure (consisting of G kernels) was never split across two processors and always followed the input or PE that resides at the $g=0$ layer.

It was found that in partitioning either the focused gamma net or gamma model, additional local processor storage space was required for the gamma weights and kernel outputs. The amount of extra storage was specifically quantified in section 4.5. In addition,

local processing time was expected to increase when compared to a similar MLP realization. However global communication was shown to remain constant when compared to the partitioned MLP network. This was due to the global communication being dependent solely on the number of PEs in the net which does not change between the static and dynamic cases. Thus a higher degree of parallelism is expected for the partitioned gamma networks executing on a parallel system. Concluding this chapter, it was decided that any type of dynamic network containing *local recurrence* could be partitioned efficiently using the backpropagation alignment scheme.

Following the gamma model algorithm analysis, a tool referred to as algorithmic timing parameter decomposition (ATPD) was applied to the MLP operating in retrieval. This was done to illustrate at a simple level how to decompose and quantify the computational time associated with executing a given algorithm on a particular architecture. Three measurements of system performance were also defined to aid in critiquing the ATPD results. They were the system's computational bandwidth, cost and order of growth in the computation time formula. Using these three measures and varying the architectures executing network retrieval, one was able to observe the changes in the ATPD cost function. Upon completing the retrieval ATPD analysis, backpropagation was then decomposed for a serial machine. Basic computation time equations were developed for each phase in the backpropagation process and then summed to yield a total computation time formula. This formula was then adapted to produce individual total computation time formulas for Neuro Turbo and Kung's systolic array ring. By observing the results of the serial and parallel system total computation time formulas, new ideas emerged in how to improve existing parallel systems. This would later lead to a system developed by our lab nicknamed the CNEL Machine.

After studying and applying ATPD to other researchers systems, discussions began to occur between my advisor and a masters student assigned to designing/building a parallel neural accelerator for CNE Lab use. The result of this being that the proposed architecture be based on a floating point digital signal processor chip (DSP). The DSP was selected because it was considered computationally flexible, inexpensive, contained a fast floating point multiple-accumulate operation, and would allow the system to be available to all members of the lab. Please consult section 6.1 for the details supporting this claim.

Having selected the DSP as the base processor, three simple parallel configurations were formulated. They are the common bus, dual bus and ring configuration systems. Total computation time formulas were then created for each architecture as was performed for Neuro Turbo and the systolic arrays. These formulas were then coded into a C program containing a NeXT Interface Builder front-end input panel. See Figure 6-4. This allowed a user to change a timing primitive, system processor count (P), or network size and instantly view the effect on the total computation time. Initial timing primitives (required in the computation time equations) were obtained by checking hardware specifications for the DSPs, local memory and dual port memory. In addition, the masters student assigned to the project ran simulations on VALID to yield bus transfer timing primitives.

At this point, it should also be mentioned that the Texas Instrument TMS320C31 was chosen as the DSP. This processor was selected for its internal parallel data paths which can be used to fetch a weight (external memory) and activation (internal memory) simultaneously. In addition, because the C31 had a pipelined ALU, the fetching of data and multiply accumulate can effectively be performed in one clock cycle. We found that with these two functions, major time contributing terms could be removed from the serial computational time equations. For further details please consult Suresh Venkumahanti's masters thesis [47].

Returning to the three basic proposed architectures, several computation time simulations were run for various size networks. From this study, two major observations were made. The first being that for large networks (100,000 weights or more) the performance differences between the three architectures were minimal. Second, for smaller networks (several hundred weights), the ring out performed the common bus and dual bus systems only by a factor of two. This improvement was also not considered important due to the relatively short training times associated with smaller networks. e.g., training taking 10-20 minutes rather than 5-10 minutes. Hence the ring was deemed too expensive to build for the limited performance improvement over the other two architectures. Instead the CNEL Machine was chosen to be based on the dual bus design. The main reason for selecting this over the common bus was for reduced bus capacitance. The two separate global buses in the dual bus architecture essentially divide the loading created by the P system processors. See Figure 6-16.

Upon completing the studies for the CNEL Machine, the author began to look for an existing platform on which the computation time formulas generated in Chapters 5 and 6 could be tested. Fortunately, at this point in the research, software became available for creating a loosely coupled coarse granular parallel system out of the NeXT Machines located in the lab and at professor's offices. This clustering package is known as PVM and a detailed description of how to initiate and use it is presented Chapter 7.

PVM was used to create a parallel system which conformed to the *common bus architectural model*. The only difference being that the globally shared memory is replaced with message passing between the nodes and a single host. PVM programs were written to train a multi-layer perceptron and a focused gamma net. During the creation of the PVM multi-layer perceptron programs, another masters student from the CNE Lab finished building and testing a NeXT Machine prototype board containing four DSP56000s.

Upon completion of debugging the hardware, the student offered to write an application for partitioning the MLP across the four DSPs according to the backpropagation alignment scheme. The author now had two systems (workstation cluster and multiple DSP board) which could be used to test computation time predictions and the overall efficiency of the backpropagation alignment partitioning scheme. However before this could proceed, timing primitives were needed to characterize both systems.

Rather than estimate primitives at the elementary operation level, primitives were estimated for software modules or subroutines corresponding to phases in the total computation time formulas. See section 8.2 for greater detail. This was performed for three main reasons. The first being that the application software was already written. Second, estimating at the subroutine level would take into account any overhead generated by writing the software in a high level language, and third it was easiest to simply time the existing routines. Thus routines were timed in both the multiple DSP and cluster system MLP applications. The times were then divided by an associated operation count (given by the computation time equations) to yield an average module timing primitive.

Once generated, the module primitives were plugged into the total computational time formula, equation (6-4), to yield predictions for the cluster and multiple DSP board. These were then compared to actuals derived a short time later. See Figure 8-3. The MLP predictions for the DSP system were found to be more accurate when compared to the workstation cluster. The differences in prediction accuracy between the two were attributed specifically to the ability to predict the global communication times for each system. In the DSP system, the global communication time was based on Motorola 68040 bus transfers while the cluster globally transferred data via Ethernet. However, it was observed that for very large networks (10^5 weights or greater), the two predictions converged at around 92-94 percent accuracy. This is due to the larger portion of local processing time in

the total computation time and the shrinking of the variance in Ethernet communication times. When large packets of data (several thousand bytes) were sent across the Ethernet channel, the associated send/receive primitives were found to stabilize. This is attributed to the communication being less sensitive to Ethernet header overhead. In a short message the header is a significant portion of the message whereas in a long message it is much less significant.

The DSP system was also shown to have a better parallel speed-up factor when compared to the cluster. See Figure 8-4. This was due the DSP system's smaller ration of global communication transfer to local processing time. However if additional memory was added to the DSP board, the speed-up factors should be equivalent between the two systems for 10^6 weights or greater.

As expected, the DSP system out performs the cluster in terms of total computation time. See Figure 8-5. *However for large networks (10^5 weights or more) the DSP based system is only 2.4 times as fast as the cluster.* This was a surprising result and is attributed to several important factors. The first is the previous mentioned point that the Ethernet transfer primitive has significantly shrunk when compared to the primitive observed for a smaller net. The second is that the 68040 (cluster processor) becomes very efficient at crunching large quantities of floating point numbers in a simple multiply-accumulate loop. This is due to the 68040's internal instruction cache and pipelined data flow to the ALU. The third and final point is the fact that the DSP56000 is a fixed point processor and therefore loses a significant amount of time in scaling operations.

After testing the MLP applications, the focused gamma net was implemented across the cluster as described in section 4.5. Primitives for new modules were again estimated as in the MLP case and predictions were generated for the total computation time. The predictions were then compared to actuals and approximately the same error

trends were observed as those for the MLP tests. Graphing the focused gamma net's speed-up curve (parallel efficiency), the curve was found to closely match the efficiencies found for the MLP testing. See Figure 8-7. However when the two algorithm speed-up curves were plotted according to the static weights found in both topologies, the Focused gamma Net was found to be more efficient. See Figure 8-8. This meant that if a user migrated from a MLP to the focused gamma net, a greater degree of parallelism was expected for nets with the same number of inputs, hidden PEs and outputs. Because of this increase in parallel efficiency, doubling the number of weights in the network does not necessarily double the computation time. This was evident by Figure 8-9. Hence one may be able to add function to a network and not increase the total computation time.

In conclusion, a method was presented for decomposing an algorithm into sums and products of system dependent timing primitives. The result is a total computation time formula derived initially for a serial machine that can be later adapted for a parallel machine. The formula can be used to predict the behavior of a system running a given algorithm. This was exemplified with the multiple DSP and cluster system studies. It was found that accurate predictions can be made for processor related operations but that estimating Ethernet times is difficult. However providing accurate predictions can be made (e.g., multiple DSP system and cluster results for large networks), the estimations can be used to optimize the system processor count for a particular parallel application (e.g., network size). For example, in both the case of the cluster and the DSP based system, it would be more advantageous to back down the system processor count P for small networks (less than 10^4 weights).

In addition, the individual components of the total computation time formula can be used to observe and improve the limitations of the parallel architecture and associated partitioning scheme. This was the case for the CNEL Machine. The Neuro Turbo system

was decomposed, analyzed and re-designed in terms of bus structure and selection of local memory and processor type.

9.2 Future Work

Due to the speed-up results found for the large network cluster tests, the lab is considering further research in clustering 66MHz PC boards. The plan is to possibly purchase up to ten of the mother boards (headless) and connect them via a dedicated Ethernet local area network for the purpose of running parallel neural network applications. Other faster global communication links are also being considered such as fiber optics or twisted pair for network training data transfer. The idea being that PVM commands (via Ethernet) would be used to control the parallel system while large chunks of data required by the nodes is sent via a faster communication mechanism. From this, one would expect much more accurate prediction results for the computation time models and better speed-up factors for smaller networks.

Presently, the CNEL Machine is undergoing the last stages of construction and test of a four processor system and is expected to be running by the end of the semester. It is the hope of the author that both the MLP and focused gamma net algorithms will be partitioned on this network. Better prediction results and speed-up factors are expected for this system when compared to the cluster and multiple DSP56000 board. Because this is a floating point processor with fast local memory, a 50+ improvement over the NeXT Machine is conservatively estimated. See section 6.8 for the details involved in obtaining this estimation. A further acceleration may be possible if the scheduling of tasks or operations is made more efficient. For example, when one system finishes computing a portion of the partial nets it has been assigned, it immediately writes them to global memory and then continues on with the rest of the partial net calculations. In other words, the task is

split into sub-tasks and a global scheduling algorithm is employed. This would enable local processing (other DSPs) time to overlap the global communication time.

Another area of potential study is how to partition an object oriented neural net program over a group of DSPs or computer cluster. A new object oriented neural net package referred to as "NeuroSolutions" has recently been installed in the lab which allows one to graphically build and probe a neural network. The lab is currently discussing how this could be partitioned and interfaced to the CNEL Machine and/or to a computer cluster. The cluster may offer the easiest implementation because an operating system (NeXT Step486) already exists which can execute objects remotely on another PC or workstation.

Future research is also required to address the question of how to partition globally recurrent networks on a parallel machine. The fully connected case, due to exhausting interconnection, will be inefficient to implement on a parallel system. However intermediate cases of recurrence have practical value, as the artificial neural network literature illustrates. These are the ones that should be studied next. The author first suggests that the process be analyzed as was done for the MLP and focused gamma net. Write the algorithm equations in a summation form, break the process into phases and develop a flow chart for the phases dependencies. Another useful tool is to create a dependency graph. In any event, having observed the parallelism in the algorithm, identify the neighborhoods of recurrence and initially apply the weight storage according to backpropagation alignment partitioning scheme. Next apply ATPD to develop a cost function for the process operating on a given architecture. In an iterative manner (and through intuition), continue making changes to the partitioning scheme and observe the new cost function.

One final area of work is in extending the use of the ATPD computation time formulas in searching decision spaces related to optimizing parallel system performance. The computation time formulas can be considered a cost function and thus employed to mea-

sure different techniques designed to search a given solution space. For example, it may be possible to obtain an optimal partitioning scheme or scheduling routine via a genetic algorithm or other search oriented techniques.

REFERENCES

- 1 D. Hammerstrom, "Neural Networks at Work," IEEE Spectrum, vol. 30, no. 6, pp. 26-32, June, 1993.
- 2 K.L. Moore, "Artificial Neural Networks: Weighing the Different Ways to Systemize Thinking," IEEE Potentials, vol. 11, no.1, pp. 23-28, February, 1992.
- 3 E. Deprit, "Implementing Recurrent Back-Propagation on the Connection Machine," IJCNN, vol. 2, pp. 295-314, June, 1989.
- 4 G. Chinn, K.A. Grajski, C. Chen, C. Kuszmaul, S. Tomboulion, "Systolic Array Implementations of Neural Nets on the MasPar MP-1 Massively Parallel Processor," IJCNN, vol. 2, pp. 169-173, June, 1990.
- 5 M. Holler, S. Tam, H. Castro, R. Benson, "An Electrically Trainable Artificial Neural Network (ETANN) with 10,240 Floating Gate Synapses," IJCNN, pp.191-196, June, 1989.
- 6 A. Iwata, Y. Yoshida, S. Matsuda, Y. Sato, N. Suzumura, "An Artificial Neural Network Accelerator using General Purpose 25 Bit Floating Point Digital Signal Processors," IJCNN, vol. 2, pp. 171-182, June, 1989.
- 7 G.M. Shepherd, Neurobiology, Oxford University Press, New York, 1983, pp. 101-119.
- 8 K. Simpson, Artificial Neural Systems, Pergamon Press, New York, 1990.
- 9 R. Hecht-Nielsen, Neurocomputing, Addison-Wesley, New York, 1990
- 10 B. Widrow, S.D. Stearns, Adaptive Signal Processing, Prentice-Hall, Englewood Cliffs, New Jersey, 1985.
- 11 R.P. Lippmann, "An Introduction to Computing with Neural Nets," IEEE ASSP Magazine, pp. 4-21, April, 1987.
- 12 M. Caudill, "Neural Networks Primer," Part III, AI Expert, pp. 17-23, March, 1988.
- 13 M. Caudill, "Neural Networks Primer," Part II, AI Expert, pp. 17-23, February, 1988.
- 14 T. Kohonen, "An Introduction to Neural Computing," Neural Networks, vol. 1, pp. 3-16, 1988.

- 15 S.Y. Kung, VLSI Array Processors, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.
- 16 S.G. Akl, Design and Analysis of Parallel Algorithms, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.
- 17 S. Thrun, F. Smeija, "A General Feed-Forward Algorithm for Gradient Descent in Connection Networks," Arbeits Papier, German National Research Center for Computer Science, Sankt Augustine, West Germany, 1991.
- 18 D.E. Rumelhart, G.E. Hinton, R.J. Williams, "Learning Representations of Back-Propagation Errors," *Nature*, vol. 323, pp. 533-536, October, 1986.
- 19 N.H. Farhat, D. Psaltis, A. Prata, "Optical Implementation of the Hopfield Model," *Applied Optics*, vol. 24, pp. 1469-1475, May, 1985.
- 20 A.D. Fisher, W.L. Lippincott, J.N. Lee, "Optical Implementations of Associative Networks with Versatile Adaptive Learning Capabilities," *Applied Optics*, vol. 26, no. 23, pp. 5039-5054, December, 1987.
- 21 T.J. Schwartz, "A Neural Chips Survey," *AI Expert*, December, pp. 34-37, 1990.
- 22 A.P. Thakoor, A. Moopenn, J. Lambe, S.K. Khanna, "Electronic Hardware Implementations of Neural Networks," *Applied Optics*, vol. 26, no. 23, pp. 5085-5091, December, 1987.
- 23 W.A. Fisher, R.J. Fujimoto, R.C. Smithson, "A Programmable Analog Neural Network Processor," *IEEE Transactions on Neural Networks*, vol. 2, no. 2, pp. 222-228, March, 1991.
- 24 A. Agranat, A. Yariv, "A New Architecture for a Microelectronic Implementation of Neural Network Models," *IEEE First International Conference on Neural Networks*, Vol. 3, pp. 483-409, June, 1987.
- 25 A. Agranat, C. Neugebauer, A. Yariv, "A CCD Based Neural Network Integrated Circuit with 64K Analog Programmable Synapses," *IJCNN*, Vol. 2, pp. 551-555, June, 1990.
- 26 M. Tomlinson, D. Walker, M. Sivilotti, "A Digital Neural Network Architecture for VLSI," *IJCNN*, Vol. 2, pp. 545-550, June, 1990.
- 27 S.C. Garth, "A Chipset for High Speed Simulation of Neural Network Systems," *IJCNN*, vol. 3, pp. 443-452, June, 1987.
- 28 T. Wantanabe, Y. Sugiyama, T. Kondo, Y. Kitamura, "Neural Network Simulation on a Massively Parallel Cellular Array Processor:AAP-2," *IJCNN*, vol. 2, pp. 155-161, June, 1989.
- 29 J. Hicklin, H. Demuth, "Modeling Neural Networks on the MPP," *IEEE 2nd Symposium on Frontiers of Massively Parallel Computation*, pp. 39-42, 1988.
- 30 J. Brown, M. Garber, S. Venable, "Artificial Neural Network on an SIMD Architecture," *IEEE 2nd Symposium on Frontiers of Massively Parallel Computation*, pp. 43-47, 1988.

- 31 J. Hwang, S.Y. Kung, "Parallel Algorithms/Architectures for Neural Networks," *Journal of VLSI Processing*, Vol. 1, pp. 221-251, 1989.
- 32 S.Y. Kung, "Tutorial: Digital Neurocomputing for Signal/Image Processing," *IEEE Conference on Neural Networks for Signal Processing*, pp. 616-644, 1991.
- 33 R. Drafz, K. Smith, E. Dowling, M. Griffin, M. Lynch, F. Taylor, "Turn a PC into a Supercomputer with Plug-In Boards," *Electronic Design*, pp. 89-93, November, 1988.
- 34 A.B. Smith, "A Parallel PDP Network Simulator," *IJCNN*, vol. 3, pp. 377-384, June, 1987.
- 35 B. DeVries, J. Principe, "The Gamma Model--A New Neural Model for Temporal Processing," *Neural Networks*, vol. 5, pp. 565-576, August, 1992.
- 36 J. Principe, B. DeVries, P. de Oliveira, "The Gamma Filter--A New Class of Adaptive IIR Filters with Restricted Feedback," *IEEE Transactions on Signal Processing*, vol. 41, no. 2, pp. 649-656, February, 1993.
- 37 A.B. DeVries, "Temporal Processing with Neural Networks--The Development of the Gamma Model," Ph.D. dissertation, Department of Electrical Engineering, University of Florida, Gainesville, 1991.
- 38 W.C. Lefebvre, "An Object Oriented Methodology for the Analysis of Artificial Neural Networks," M.S. thesis, Department of Electrical Engineering, University of Florida, Gainesville, 1992.
- 39 S.Y. Su, Database Computers, McGraw-Hill Book Company, New York, 1988.
- 40 E.V. Krishnamurthy, Parallel Processing Principles and Practice, Addison-Wesley, New York, 1989.
- 41 Motorola Inc., "DSP56000 Fixed Point Digital Signal Processor User's Manual," Phoenix, Arizona, 1989.
- 42 Texas Instruments, "TMS320C25 User's Guide," Dallas, 1986.
- 43 V.S. Sunderam, A. Beguelin, J. Dongarra, A. Geist, R. Manchek, "A User's Guide to PVM Parallel Virtual Machine," U.S. Department of Energy, ORNL/TM-11826, July, 1991.
- 44 V.S. Sunderam, "PVM: A Framework for Parallel Distributed Computing," U.S. Department of Energy, Contract DEAC05-84OR21400, Washington, D.C., July, 1991.
- 45 G.A. Geist, V.S. Sunderam, "Network Based Concurrent Computing on the PVM System," U.S. Department of Energy, Contract DEAC05-84OR21400, Washington, D.C., July, 1991.
- 46 T. C. Hornquist, "A Multi-DSP Parallel Processor for the NeXT Workstation with Application to Neural Networks," M.E. thesis, Department of Electrical Engineering, University of Florida, Gainesville, 1993.

- 47 S. Venkumahanti, "A DSP Based MIMD Machine for Neural Network Acceleration," M.S. thesis, Department of Electrical Engineering, University of Florida, Gainesville, 1994.

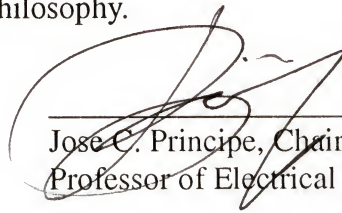
BIOGRAPHICAL SKETCH

Karl Stewart Gugel was born on October 31, 1959, in Midland, Michigan, to Carl Jack Gugel and Yvonne Florine Gugel.

He obtained his undergraduate degree in electrical engineering from Michigan Technological University at Houghton, Michigan, in 1981. Upon graduation he went to work for the IBM Corporation in Boca Raton, Florida. In 1987, under IBM's Graduate Work Study Program, Karl obtained a Master of Science in engineering degree from Florida Atlantic University in Boca Raton.

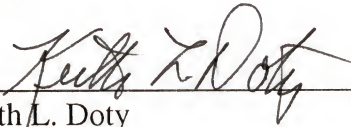
In January of 1989 he took an educational leave of absence and entered the Computational NeuroEngineering Laboratory at the University of Florida. Since then he has been working on partitioning artificial neural networks onto coarse granular parallel systems under the supervision of Dr. Jose C. Principe.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



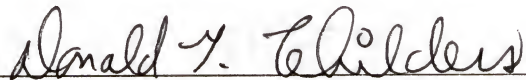
Jose C. Principe, Chairman
Professor of Electrical Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



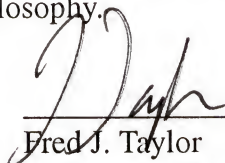
Keith L. Doty
Professor of Electrical Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Donald G. Childers
Professor of Electrical Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Fred J. Taylor
Professor of Electrical Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



John M.M. Anderson
Assistant Professor of Electrical Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Michel A. Lynch

Lecturer of Electrical Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Richard E. Newman-Wolfe

Assistant Professor of Computer and
Information Sciences

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

December 1993



Winfred M. Phillips

Dean, College of Engineering



Karen A. Holbrook

Dean, Graduate School